Distributed Computing Framework Based on Software Containers for Heterogeneous Embedded Devices

Daniel José Bruzual Balzan

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 9.10.2017

Supervisors

Mario Di Francesco, PhD.

Yannis Velegrakis, PhD.

Advisor

Mario Di Francesco, PhD.

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Daniel José Bruzual Balzan

Title Distributed Computing Framework Based on Software Containers for Heterogeneous Embedded Devices

Degree programme ICT Innovation

Major Service Design and Engineering **Code of major** SCI3022

Supervisors Mario Di Francesco, PhD., Yannis Velegrakis, PhD.

Advisor Mario Di Francesco, PhD.

Date 9.10.2017 **Number of pages** 58 **Language** English

Abstract

The Internet of Things (IoT) is represented by millions of everyday objects enhanced with sensing and actuation capabilities that are connected to the Internet. Traditional approaches for IoT applications involve sending data to cloud servers for processing and storage, and then relaying commands back to devices. However, this approach is no longer feasible due to the rapid growth of IoT in the network: the vast amount of devices causes congestion; latency and security requirements demand that data is processed close to the devices that produce and consume it; and the processing and storage resources of devices remain underutilized. Fog Computing has emerged as a new paradigm where multiple end-devices form a shared pool of resources where distributed applications are deployed, taking advantage of local capabilities. These devices are highly heterogeneous, with varying hardware and software platforms. They are also resource-constrained, with limited availability of processing and storage resources. Realizing the Fog requires a software framework that simplifies the deployment of distributed applications, while at the same time overcoming these constraints. In Cloud-based deployments, software containers provide a lightweight solution to simplify the deployment of distributed applications. However, Cloud hardware is mostly homogeneous and abundant in resources. This work establishes the feasibility of using Docker Swarm – an existing container-based software framework – for the deployment of distributed applications on IoT devices. This is realized with the use of custom tools to enable minimal-size applications compatible with heterogeneous devices; automatic configuration and formation of device Fog; remote management and provisioning of devices. The proposed framework has significant advantages over the state of the art, namely, it supports Fog-based distributed applications, it overcomes device heterogeneity and it simplifies device initialization.

Keywords Internet of Things, Software Containers, Fog Computing, Lightweight Virtualization, Embedded Devices, Docker Swarm

Contents

Abbreviations and Acronyms

API	Application Programming Interface
DNS	Domain Name System
IoT	Internet of Things
IP	Internet Protocol
IPVS	IP Virtual Server
JSON	JavaScript Object Notation
LAN	Local Area Network
MAC	Media Access Control
NAT	Network Access Translation
REST	Representational State Transfer
SSH	Secure Shell
TCP	Transmission Control Protocol
UI	User Interface
YAML	YAML Aint́ Markup Language
VETH	Virtual Ethernet Device
VM	Virtual Machine
VPN	Virtual Private Network
VTEP	VXLAN Tunnel Endpoint
VXLAN	Virtual Extensible LAN

1 Introduction

The Internet has radically transformed many aspects of human society, such as culture, economy, governance and more. The presence of systems connected online is apparent in our daily life: to connect with friends and relatives, to buy goods online, to reach business arrangements through emails or to handle paperwork, the Internet has become an indispensable tool for a large part of the human population.

Whereas in the year 2000 only 6.5 percent of the world population had access to the Internet, this number grew to 48 percent in 2017 [46] and is expected to keep growing. However, a new paradigm is emerging where machines become the main generators and consumers of data and communicate through the Internet directly with each other, with little or no human intervention. This paradigm is referred to as the Internet of Things.

In the Internet of Things everyday objects are equipped with sensors, actuators and communication capabilities. These devices are present in homes, city infrastructure, buildings, mines and factories and in nearly any other place conceivable, and therefore are highly heterogeneous. They monitor and act on their environment. They bridge the gap between the physical and digital worlds. They communicate among themselves and are able to reach decisions and modify their behavior dynamically. They are connected with external systems where data can be analyzed and be used to continuously improve their performance [29].

Not only has Internet become more widespread, but hardware has also become smaller, more efficient and cheaper [16]. In fact, many embedded IoT devices posses more powerful configurations than most computers did a decade ago. For this reason, it makes sense to take advantage of local processing resources on these devices. Traditional approaches in software development for online systems involved offloading data to powerful remote servers in the cloud, which performed computations and decision-making and then relayed back the results to end-devices [41, 25]. However, in the IoT this option is not always valid, as devices interact in performance critical scenarios or with intermittent or expensive connectivity. For these reasons, new approaches have emerged, that take advantage of local resources to perform computations, resulting in lower latency and less bandwidth utilization.

One of such paradigms is Fog Computing [48]. Multiple connected end-devices pool their resources and process computations and storage in their locality, instead of entirely relying on the cloud. This presents the challenge of developing a software framework for deploying and managing applications on clustered devices: a framework that is generic enough to handle the multiple and varied application scenarios of the IoT, and that is compatible with the high heterogeneity and variety of connected devices.

In cloud-based systems virtualization technologies have been vastly employed for similar purposes: large groups of computer servers are grouped into clusters that can managed as a pool of resources. Virtual machines are portable virtual environments that encompass applications as well as all of their requirements and can be allocated across the cluster, abstracting the underlying hardware. Several frameworks have been developed to manage and deploy these portable environments, enabling multiple

applications to interact and quickly adapting to changes in requirements and failures, such as hardware and software crashes.

Recently, lightweight virtualization based on software containers has gained tremendous popularity for cloud-based deployments [39]. Containers are portable packages that include an application and all the libraries required to run it. Containers are similar to virtual machines; however, virtual machine images include entire operating systems, whereas containers re-use the host operating system. For this reason they are much smaller in size and require less runtime resources, thus making them suitable for resource-constrained environments [45].

One of the most popular container-based solutions is Docker, which includes a whole set of tools for generating, deploying and managing containers [4]. It has become popular due to its support for multiple hardware and operating system platforms and as it greatly simplifies the whole container lifecycle. Docker also provides a feature called Docker Swarm, which allows creating a cluster of devices where distributed container-based applications are deployed.

Docker is compatible with Linux-based operating systems, like those that are found in many embedded devices and runs as an additional software layer, without requiring the installation of custom operating systems. Containers provide portable virtual environments capable of abstracting the underlying heterogeneity. Moreover, containers are lightweight, and therefore compatible with the resource-constrained nature of these devices. Docker Swarm provides tools for management and provisioning of clustered devices.

1.1 Scope and Goals

This research targets developing a distributed computing framework for heterogeneous IoT devices, i.e., Fog Computing. Any framework suitable for IoT deployments must: be easy to set up; support heterogeneous and resource-constrained devices; allow deploying applications which interact directly in the fog; and provide a management interface which allows remote provisioning of devices [44]. This research establishes the feasibility of Docker Swarm as a suitable framework, by developing custom tools to satisfy the above-mentioned requirements.

The research topic is broken down into the four research goals detailed below.

- *Support for running distributed applications on the managed devices.* The developed framework must support a multi-tenant model where multiple applications can make use of available resources without affecting each other. However, running applications should also be able to communicate among themselves when required.

- *Support for heterogeneous and resource-constrained devices.* Devices are heterogeneous in their hardware resources and also in the availability of sensors and actuators. The framework should allow modeling such a heterogeneity and using it when required; moreover, it should also be capable of abstracting heterogeneity and deploying applications regardless of the underlying platform-

specific details. Applications should also be compact so that they can be quickly deployed onto devices so as to save bandwidth resources and storage space.

- *Simplify the set up and the initial configuration of connected devices.* The developed framework should be scalable to support the large number of devices that make up IoT deployments. Manual configuration of devices should be avoided and replaced with automated methods.

- *Support remote provisioning and management of IoT devices.* Applications should be easily deployed and updated without requiring physical access to devices. Stakeholders should also be able to monitor the operating status of managed devices through some graphical user interface.

1.2 Contribution

The contributions of this work are the following:

- Establishing the feasibility of Docker Swarm as a framework for distributed computing on IoT devices; in particular, how to use existing features to model device heterogeneity, to deploy multiple applications per device, to deploy applications across heterogeneous devices, and to enable deployed applications to interact.

- Developing a technique to build compact container images that support heterogeneous devices and incur in little overhead.

- Designing a configuration server which automates the initial formation of a cluster of devices, thus eliminating the need of manual intervention.

- Enabling the remote management and provisioning of devices connected through Docker Swarm, by means of a cloud-based management user-interface.

1.3 Structure

The rest of this work is organized as follows. Chapter 2 introduces the main concepts about system virtualization, starting with traditional hypervisor-based virtual machines and continuing with container-based virtualization, with focus on Docker and Docker Swarm. Chapter 3 introduces the Internet of Things and a background on different computational models that have been applied. Chapter 4 describes the design and implementation of a container-based framework for distributed computing on IoT devices. Chapter 5 provides an evaluation of this framework. Finally, Chapter 6 concludes this work and presents some future research directions.

2 System Virtualization

In cloud computing environments, system virtualization has become the standard model that service providers employ to manage customer applications. It allows providers to serve multiple customers by adopting a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [18]. Virtual environments can be deployed or re-deployed in different physical machines in a way completely transparent to the customer.

System virtualization involves an encapsulating software layer that surrounds an operating system, which provides the same inputs, outputs and behavior that would be expected from the physical hardware [40]. This technology allows splitting a physical computer into several isolated virtual environments, each with its own configuration and applications.

The prevalent virtualization technologies for Linux are *hypervisor-based virtualization* and *container-based virtualization*.

2.1 Hypervisor-based Virtualization

A hypervisor or Virtual Machine Monitor (VMM) is a piece of software executed on a physical computer with privileged control of hardware and software resources. It enables splitting and allocating locally-available *host* resources (e.g., memory, CPU) into a collection of *guest* virtual machines (VM), which provide an identical execution environment to that of the real machine [42]. There are two prevailing hypervisor-based architectures (Figure 1). Type 1 hypervisors run directly on top of the hardware and use special built-in instructions of the CPU to execute virtual machines; Type 2 hypervisors run as an additional software layer on top of the host operating system.

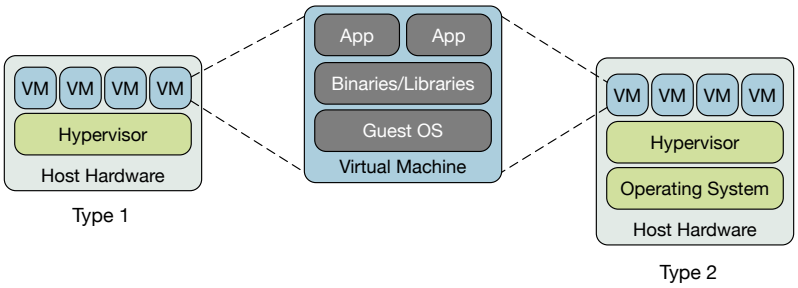

Figure 1: Hypervisor-based virtualization architectures

Several competing hypervisor-based solutions have been developed, such as KVM, Microsoft Hyper-V Server, and Oracle VMServer. Although they each exhibit optimizations for certain hardware or deployment scenarios, they share the following features:

- *Transparency:* Guest operating systems and software can be run within the virtual machine environment without any modification. In fact, the operating

system and the VM are unaware that they are in a virtual environment. Hence, any program that is run inside a virtual machine produces the same output as if it were run directly on the real machine.

- *Isolation:* Distinct virtual machines can be run on the same physical host system, isolated from each other. Each virtual machine has its own version of software running inside it, including operating system. Software failure inside a virtual machine affects no other virtual machines or the host operating system. Similarly, the performance of a virtual machine is not degraded by another resource-intensive VM.

- *Encapsulation:* The files that make up a virtual machine are contained in a virtual hard disk. This disk can be moved from one host system to another, backed-up or copied. This way, a virtual machine can be easily replicated or migrated to another server.

- *Manageability:* The hypervisor allows controlling the execution of VMs and their available resources through a management interface.

The main advantages of hypervisor-based approaches stem from near-complete isolation between guest operating systems. This guarantees that the damage induced by a bug, crash, or security breach remains contained to the VM where they originate. Moreover, each VM has its own operating system and varied operating system types (e.g., Linux, Unix, Windows) can coexist in the same physical hardware. This enables service providers to cater a wide scope of consumer's needs with limited hardware.

However, hypervisor-based solutions have drawbacks, mainly related to resource utilization. Firstly, in case several instances rely on the same operating system (e.g., Windows Server 2016), it is replicated multiple times in the server, causing unnecessary bloating. The same applies for common runtime libraries and programs. Secondly, starting a Virtual Machine requires booting up an operating system, which results in additional startup time. Finally, the hypervisor adds runtime overhead as it replaces privileged instructions executed inside the guest VMs.

2.2 Container-based Virtualization

Container-based virtualization provides a lightweight alternative to hypervisor-based virtualization. Virtualization is achieved at the level of the host operating system, without a hypervisor, thus reducing runtime overhead (Figure 2). Virtual environments (*containers*) all share the same operating system as the host, reducing storage overhead.

A container image is a stand-alone package from which containers can be instantiated and executed. It encapsulates everything it requires to run: code, libraries, binaries, and runtimes, as well as application- and system-specific configuration [14]. This bundle can be distributed, backed-up, or copied. Multiple instances can be executed in the same physical server or different servers, to provide fault-tolerance.

Processes belonging to different containers are all executed side-by-side on the host operating system. However, through features of the operating system kernel,

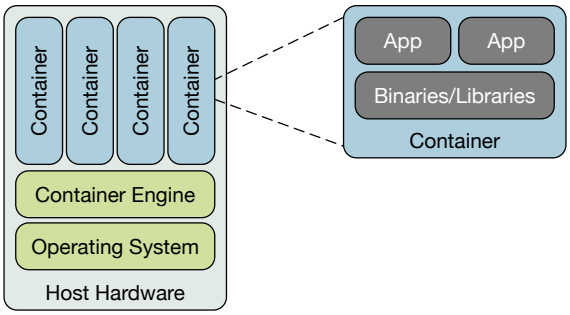

Figure 2: Hypervisor-based virtualization architectures

they have different views of the system: network interfaces, process IDs, interprocess communication, and directory trees. In this way, containers remain isolated from each other, as if running on different physical machines.

Although container-based virtualization can be implemented in any operating system, most popular solutions (e.g., Docker, rkt) rely on features of the Linux kernel. They provide a management interface on top of these features, enabling the execution, administration and monitoring of containers. Resource management is achieved through Control Groups (cgroups), which restrict resource usage per process groups. Cgroups make it possible to limit or prioritize CPU, memory and I/O for different containers. Isolation is achieved through the use of namespaces, which allow different processes to have different views of the system.

In the context of cloud computing, the resources saved by reusing the same base operating system enable a higher number of parallel deployments per physical machine. Moreover, as container processes execute directly on the host machine they execute transparently and with minimal overhead. Additionally, starting or restarting a container is a fast operation, given that starting a process is much faster than booting up an operating system.

In the following subsections the enabling technologies of container-based virtualization are further explained.

2.2.1 Control Groups

Control Groups, or *cgroups*, allow the allocation of resources — such as CPU time, system memory, network bandwidth, or combinations of these resources — among user-defined groups of processes running on a system [2]. They allow fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. Dynamic reconfiguration of resources on a running system is also possible.

Cgroups are organized hierarchically, and child cgroups inherit some of the attributes of their parents. Many different hierarchies of cgroups can exist simultaneously on a system. Each hierarchy is attached to one or more subsystems. A subsystem represents a single resource, such as CPU time, memory, IO operations, physical device access, network traffic, among others.

2.2.2 Namespaces

A namespace abstracts a global system resource in such a way that it appears as an isolated instance to the processes within that namespace [10]. Changes to the resource that take place inside a namespace do not affect the global resource, and viceversa. In this way, groups of processes can have different views of the same physical system, as if they were in fact running on different physical systems.

Seven types of namespaces currently implemented in the Linux kernel:

- *Mount namespaces* provide isolation of the list of mount points in each namespace instance. The processes in each of the mount namespace instances sees distinct single-directory hierarchies, possibly with different access permissions.

- *Network namespaces* provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewalls, port numbers (sockets), and so on. A physical network device can belong to exactly one network namespace, since otherwise isolation between namespaces could not be achieved. However, virtual network devices are employed to give each namespace its own network interfaces, while enabling communications with other namespaces and with external hosts. A virtual network device (VETH) pair provides a connection between a namespace and the root namespace. Internet access can be provided to the namespace by setting up routing rules and IP addresses for the host VETH interface and enabling a NAT. Communication between namespaces can be achieved by connecting host-side VETH interfaces to a bridge interface, which is equivalent to physical machines connected to an Ethernet switch.

- *User namespaces* isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keyring, and capabilities. Processes running inside a user namespace are visible from the root namespace. However, user namespaces enable these processes to have different user IDs and group IDs inside and outside the user namespace. For instance, a process can have a normal unprivileged user ID when seen from the host system while at the same time have a root ID inside the user namespace. In other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

- *PID namespaces* isolate the process ID number space. Process IDs inside a namespace are unique, but processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the processes inside the container maintain the same PIDs, and no conflict arises.

- *IPC namespaces* isolate certain Inter-Process Communication resources, namely, System V IPC [11] objects and POSIX message queues.

- *UTS namespaces* provide isolation of two system identifiers: the hostname and the NIS (Network Information Service) domain name.
- *Cgroup namespaces* virtualize the view of cgroups related to a set of processes. Each cgroup namespace has its own set of cgroup root directories.

2.3 Docker

Docker Engine is an open-source platform for developing, shipping, and running container-based applications. It builds on top of Linux cgroups and namespaces, to streamline the whole process of running containers. It provides a mechanism for packaging applications, their runtimes, and their required configurations into images; as well as the tooling for handling image versions, deploying containers, and monitoring/managing running containers. In addition to the benefits from leveraging container-based virtualization, Docker also simplifies the software release lifecycle by ensuring that the development, testing and deployments environments are the same, which is particularly useful for continuous integration and continuous development (CI/CD) workflows [4].

A container is typically stateless and ephemeral, in the sense that it can be stopped/destroyed and a new one built and put in place with minimal set-up and configuration. According to best practices [1], each container should handle a single concern. That is, it should provide the functionality of a single part of a bigger system. By decoupling large applications into multiple containers (e.g., data-store, frontend, backend), their individual components are more easily debugged and updated, and it is easier to scale horizontally only those components that need so. The portability and lightweight nature of containers also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

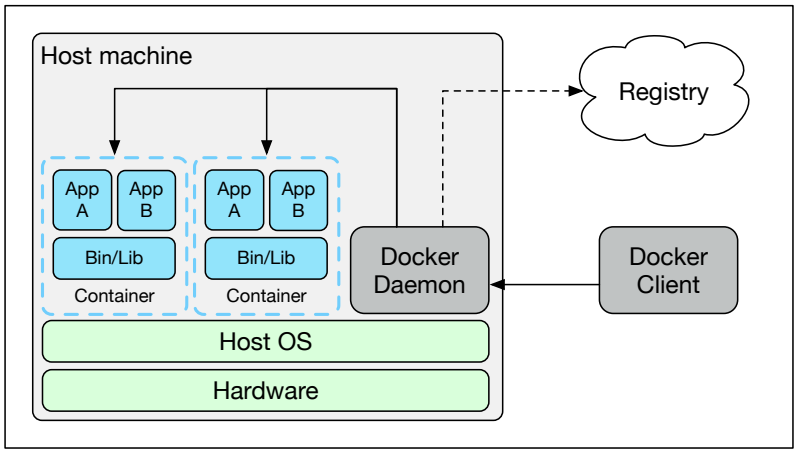

Figure 3: Docker Engine architecture

Docker Engine is built according to a client-server model (Fig. 3). The Docker daemon (`dockerd`) runs on the host system and manages containers and images. It

takes care of monitoring the status of the system, enforcing system policies (e.g., restarting failed containers), receiving and executing client commands, and creating the namespace environments where containers are run.

The client tool (`docker`) connects to a docker daemon, which can be running locally or remotely. It is a command-line tool that provides operations related to managing images (e.g., build, push, pull, update), containers (e.g., run, pause, resume, stop, remove), networks, services, swarms and Docker Engine configuration.

2.3.1 Docker Image

A Docker image is a collection of tar packages, called layers, that when extracted and combined in the correct order make up the filesystem that is reproduced inside a container. This filesystem includes all system binaries, libraries, configuration files and the user application code or program needed to create the container. There is one additional layer, that does not belong to the filesystem, but rather holds container configuration such as environment variables, and ports that should be exposed external to the container.

The layers that make up the image are read-only and can be seen as a stack (Fig. 4). When new files are added to the image or existing files are modified, the changes are written in a new layer that is placed on top of the existing layers. The resulting stack can be saved as a new image. In this way it is possible to generate new images by extending existing ones. In the figure, 4 layers are added on top of the `ubuntu:15.04` image, which in turn is constituted by its own layers (not depicted). If multiple images have common base layers, a single copy on disk of the common layers can be kept on disk and be shared by them, reducing storage space required. Layers are identified by a digest of their contents, making it easy to verify if they are already present in the host system by comparing this value against the stored ones.

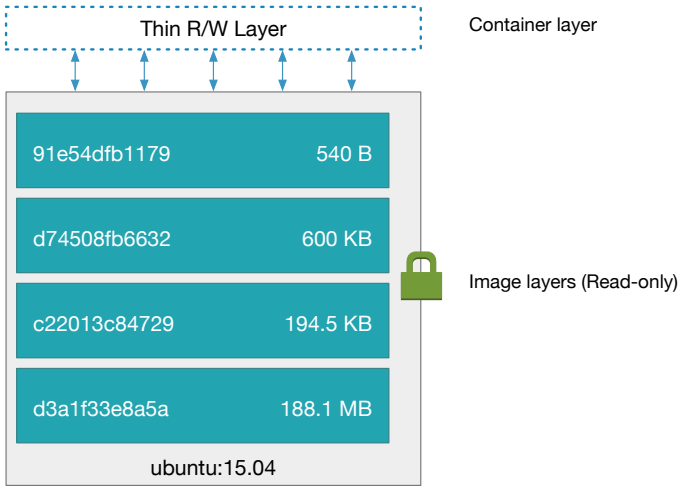

Figure 4: Representation of a Docker image as a stack of read-only layers

Images are based around specific system architectures. Many of the programs that make up the image, such as the shell, command binaries, and libraries are

compiled for a given architecture. Since lightweight virtualization employs the kernel of the host, it is required that the image architecture matches the host architecture. There is a workaround for developing images that support multiple architectures, which is described in Section 2.3.6.

Images are instantiated to create containers, they are described and generated through Dockerfiles and stored and distributed in Registries. Next, these concepts are explained.

2.3.2 Docker Container

In Docker, a container is an instance of an image along with allocated system resources. Whenever a container is started, a writable layer is created on top of the image. Any changes that occur to the filesystem are placed in this layer, while the image remains immutable. Through this mechanism, several containers can share the same image (Fig. 5). The enabling technology behind this approach is called Copy-on-Write and is provided by the storage drivers that Docker Engine employs (e.g., aufs, zfs, btrfs). Files belonging to the image layers are read-only, and since reads are non-conflicting, many instances can read from the same file. As soon as a container modifies (i.e., writes) one of these files, a new copy is made and placed in the writable layer instead of modifying the original file. Any further reads or writes of this file take place on the new version.

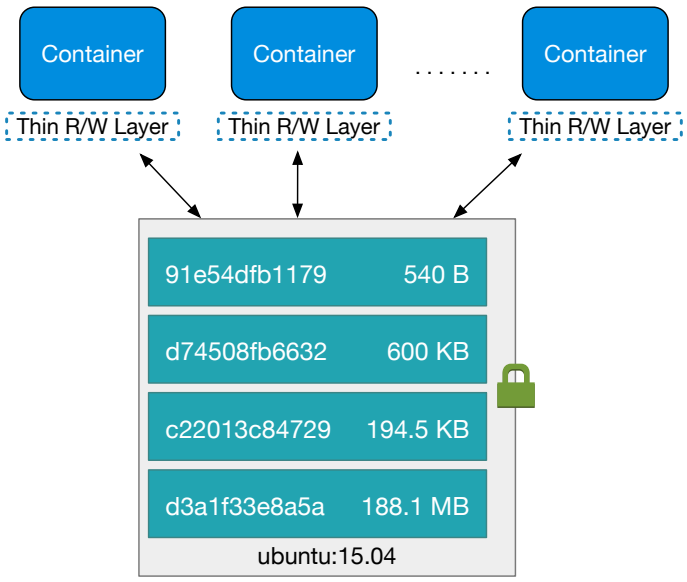

Figure 5: Multiple containers can use the same image

Docker Engine provides an API and command-line tools for starting, stopping, resuming and removing containers. When a container is started, it is possible to specify additional configuration that determine how the container interacts with the host system. For instance, ports can be published by providing a port mapping between an external and an internal port number. Other hosts on the same physical

network are able to communicate with the container through the external port, and the connection is mapped to the internal port inside the container. Volumes can also be mounted, which allow the container to read and write directories of the host filesystem. Changes to these files are persistent, even in the event that the container is stopped.

One of the advantages of containers is their isolation. They are executed as closed environments that have no side effects on the host system. However, for some applications access to parts of the host is required. For instance, in some IoT applications, containers require reading sensor values or activating actuators. For this purpose containers can be initialized with additional system capabilities that range from managing system resources, to accessing the system hardware.

2.3.3 Dockerfile

The Dockerfile is a text file that provides the Docker Engine all the information on how to build an image, configure the container and execute the contained applications. It is defined as a series of instructions that are parsed and executed line-by-line. Some of these instructions result in changes to the files inside the image, in which case they are placed on a new layer according to the layered-approach described previously. Some instructions describe system configurations, such as environment variables and network ports where the container listens for connections at runtime, and are written to a JSON configuration file, which is stored in a separate layer.

A sample Dockerfile for a web application is shown in Listing 1. A Dockerfile always starts with the `FROM` directive, which specifies the base image to build upon. If this image is not locally available, Docker Engine looks for it among its known repositories and downloads it. The `SCRATCH` base image may also be employed, which instructs Docker Engine to use an empty filesystem as the base.

```
1 FROM ubuntu:15.04
2 COPY . /app
3 RUN apt-get update && apt-get install nodejs npm
4 RUN make /app
5 EXPOSE 8080
6 ENV PORT 8080
7 CMD node /app/app.js
```

Listing 1: Sample Dockerfile for a Node.js web application

The next lines in the Dockerfile prepare the image by including files, executing shell commands, or setting configuration options. Files are copied from the host filesystem into the image with the `COPY` instruction. Similarly, remote files are fetched with `ADD`. The working directory is set with `WORKDIR`. Shell commands are run by placing the command after a `RUN` instruction. Any call to programs available inside the container is valid (e.g., using the package manager, compiling the application). Environment variables are set with the `ENV` instruction and network ports are opened for listening with `EXPOSE`.

Optionally, a single `CMD` instruction is specified. This calls scripts or programs, and is executed at runtime, whenever a container of the image is instantiated. In the

sample listing, it is used to start the application that listens for client requests on port `8080`.

There are other instructions, such as for setting the user name and group (`USER`) and performing periodic container health checks (`HEALTHCHECK`). However, the ones that have been described here are the main ones.

2.3.4 Multi-Stage Dockerfile

Dockerfiles sometimes result in bulky images as many of the tools needed for building an application, such as compilers, binaries and package managers, which are not required for its deployment, remain included as a part of the final image. For instance, the image produced by the Dockerfile in Listing 2 includes the Go compiler, although it is not needed at runtime. In the context of resource-constrained devices and expensive or limited-bandwidth connections, this overhead is by no means negligible.

```
FROM armhf/golang

COPY server.go .

RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm go build -a -ldflags '-s' server.go

EXPOSE 3333

CMD ["./server"]
```

Listing 2: Simple Dockerfile that includes compiler into final image

Multi-stage Dockerfiles allow producing images without the components needed for the build-up process. The Dockerfile is composed of different sections, called *stages*. Each stage, except the last one, constitutes an intermediate image that can be used for building the application, or for producing artifacts (e.g., dependencies) required to run it. Files resulting from one stage can be copied to successive stages, yet only files and configurations present in the final stage are part of the produced image [13].

```
FROM golang as compiler
COPY server.go .
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm go build -a -ldflags '-s' server.go

FROM scratch
COPY --from=compiler /go/server .
EXPOSE 3333
CMD ["./server"]
```

Listing 3: Multi-stage Dockerfile that cross-compiles for ARM and produces minimal image

This approach allows building compact images. In the case of compiled programs, the intermediate stages include tools such as libraries and compilers, and execute the commands that compile the application. In contrast, the final image contains only the resulting executable binary. In the case of interpreted languages, the intermediate

stages are used to compile the interpreter and fetch dependencies, which are included in the resulting image along with the application code.

2.3.5 Registry

A Docker Registry is a server-side application for storing and distributing Docker images. Images stored in a registry are identified by their name. Several versions of an image can be stored under the same name, by including a *tag* which indicates version number or name (e.g., `ubuntu:15.04`, `debian:wheezy`). The registry provides an API that can be used for querying, pushing (storing) and pulling (retrieving) images [6]. Docker Engine interacts with any registry via the API, so devices are able to pull images as they need them.

There are public, free-to-use registries, such as Docker Hub [3], where a myriad of open-source images can be found. However, for most enterprise applications it makes sense to have a private registry, which enforces security policies and which can be placed closer to the end-devices. The Docker Registry project is open-source, so anyone can host their own. Several cloud providers such as Google and Microsoft Azure provide private container registries as well.

2.3.6 Image Manifest

Registries respond to queries for specific images with a JSON response that follows an Image Manifest Schema. The latest version is the *Image Manifest Schema Version 2, Schema 2*. Listing 4 shows an image described as an array of layers (lines 9-24) and a configuration layer (lines 4-7). Additional API calls containing the content digests are made to download each layer file.

```
{
    "schemaVersion": 2,
    "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
    "config": {
        "mediaType": "application/vnd.docker.container.image.v1+json",
        "size": 7023,
        "digest": "sha256:
            b5b2b2c507a0944348e0303114d8d93aaaa081732b86451d9bce1f432a537bc7"
    },
    "layers": [
        {
            "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
            "size": 32654,
            "digest": "sha256:
                e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f"
        },
        {
            "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
            "size": 16724,
            "digest": "sha256:3
                c3a4604a545cdc127456d94e421cd355bca5b528f4a9c1905b15da2eb4a4c6b"
        },
        {
            "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
            "size": 73109,
            "digest": "sha256:
                ec4b8955958665577945c89419d1af06b5f7636b4ac3da7f12184802ad867736"
        }
    ]
}
```

Listing 4: Sample image manifest in JSON format

A feature of this Schema is that it supports multi-architecture images, by employing *fat manifests*. Fat manifests allow a registry to hold multiple images for different architectures under the same name and version tag. When a device performs a query for a given image name and version, the registry returns a manifest listing of all available architectures and the corresponding image digests. This can be seen in Listing 5, which supports two architectures. The device then downloads the image matching its architecture. In the absence of a match, one is randomly selected, leading to an initialization error.

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
  "manifests": [
    {
      "mediaType": "application/vnd.docker.image.manifest.v2+json",
      "size": 7143,
      "digest": "sha256:
          e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab815ad7fc331f",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux",
      }
    },
    {
      "mediaType": "application/vnd.docker.image.manifest.v2+json",
      "size": 7682,
      "digest": "sha256:5
          b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e94a4333501270",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ]
}
```

Listing 5: Multi-architecture (fat) manifest which supports two architectures (i.e., ppc64le, amd64)

2.4 Container Orchestration Frameworks

A server cluster is a set of connected servers that work together as a single pool of computing and storage resources, providing high availability, load-balancing, and parallel processing [26]. Managing a cluster implies keeping track of host health, deploying and monitoring applications, and providing mechanisms for distributing processing across several physical hosts. For this purpose, a software layer is employed which abstracts the complexity of several hosts and effectively presents the cluster as a single pool of resources.

Container orchestration frameworks provide such an abstraction and allow managing a cluster and deploying containers on it. These frameworks simplify all aspects of container management from initial placement, scheduling and deployment to steady-state activities such as update, health monitoring functions, and scaling. By abstracting the physical host hardware, containers moved from one host to another or scaled without affecting availability.

Container orchestration frameworks allow creating virtual networks between containers, so that they can interact regardless of the actual physical configuration. Even if two containers are running in the same machine, they interact with each other as two separate physical machines. Some frameworks also provide built-in mechanisms for service discovery and load-balancing. In this way the cluster can be seen as a truly unified pool of resources.

Recently, several container orchestration frameworks have emerged. Amongst the

most popular are Apache Mesos[1], and Kubernetes[2] which is promoted by Google. These are very flexible frameworks that allow combining several clouds, with varying underlying technologies, into a single cluster. Due to their flexibility they typically require extensive configuration. An emerging alternative is Docker Swarm, which is built-in Docker Engine, greatly reducing configuration and simplifying initialization and management of containers on a cluster. Nonetheless, all container orchestration frameworks share some common features:

- **Service declaration:** Applications that are deployed on the cluster are declared as services. This declaration, in the form of a file or a command, specifies all configuration and deployment requirements of the application, for instance, the container images to use, number of instances that should be created, placement policies, network configurations, health parameters such as redundancy and so on. In other words, the service declaration defines a target state for the application, which the orchestration framework translates into operations on the cluster.

- **Placement:** Placement refers to requirements on how containers should be distributed across the cluster. Placement policies are rules enforced by the orchestrator that specify parameters such as the target number of container instances to be created, the maximum number of instances per hosts, and the characteristics of hosts where they can be deployed (e.g., processor and memory requirements).

- **Provisioning:** Provisioning refers to initializing and executing containers on a host. The orchestration framework translates service declarations into operations on the cluster and provisions containers on the required hosts. As the cluster state changes, the framework allocates or de-allocates containers to maintain the specified state. The framework runs a software layer on each host that allows it to monitor the status of the whole cluster and instruct any host to execute a container.

- **Discovery:** Discovery refers to the mechanisms that allow applications deployed on a cluster to reach each other without knowledge of the underlying cluster configuration. This is important as containers can be allocated on varying hosts, and assigned different IP addresses every time. The orchestrator framework abstracts the actual host configuration and provides a mechanism for containers to be discovered. For instance, Docker Swarm allows interacting with services by service name, which it translates into the IP address of a container corresponding to that service.

- **Monitoring:** Orchestration frameworks track the health of the system's hosts and containers. They employ this information allocate or re-allocate containers to meet service requirements. The frameworks also allow visualizing the cluster status and managing hosts from a centralized point, typically the master host.

[1] http://mesos.apache.org/
[2] https://kubernetes.io/

2.5 Docker Swarm

Docker Swarm is a cluster management and orchestration framework for hosts running Docker Engine [12]. It is built into Docker Engine as of version 1.12, and requires minimal configuration in order to set up. As long as hosts are reachable via the network, a swarm can be set-up with few commands. Docker Swarm abstracts many of the network configurations necessary for enabling container communication on the cluster, also referred to as swarm. The instructions for deploying and configuring containers on a swarm are similar to those for deploying and configuring containers on a single host. The tools for doing so are the same, namely the Docker Engine command line interface and the Docker Engine API. For these reasons Docker Swarm is simpler to deploy, to grasp and to use than other available orchestration frameworks. Moreover, the close relation between Docker Engine and Docker Swarm guarantees good interoperability, which is not always the case with tools developed by different companies.

In a swarm hosts are referred to as a nodes, which can take management or worker roles. Instead of deploying single containers, services are deployed on a swarm. These services result in containers being run on multiple hosts. Networks allow connecting these containers in a way that they can communicate regardless of the physical host where they run. Mechanisms built into Docker Swarm enable service discovery by name and load-balancing. Next, these concepts are expanded.

2.5.1 Nodes

Hosts participating in a swarm are referred to as nodes and take manager or worker roles. Managers monitor the swarm state and orchestrate containers, whereas workers received commands from the managers and run containers. By default, managers are also workers and are capable or executing containers. Multiple managers can participate in a swarm, in which case they maintain a consolidate view of the cluster by exchanging control information (Fig. 6). In case the main manager fails, another one can replace it. The choice of node role is performed by the operator of the cluster.

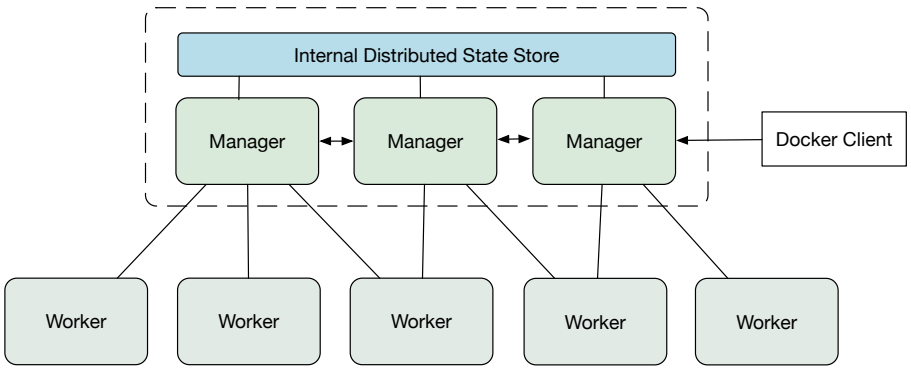

Figure 6: Architecture of a Swarm

A swarm is initiated by running a specific command on a node. Thereafter, that

node becomes the main manager (i.e., leader) of that swarm. It generates two tokens, one for workers and one for managers, that other nodes attempting to join the swarm must provide. A joining node issues a command with the token and the IP address of the manager. After the node is part of the swarm, the containers can be orchestrated on it. The orchestration is realized through a scheduler that runs on the leader, which determines in which nodes to run containers and issues commands to worker nodes.

Nodes have an availability status that indicates whether they are capable of deploying new containers. The three values that the status takes are Active, Pause, and Drain. Nodes in *Active* status are available for being scheduled to run containers. Nodes in *Pause* status cannot be scheduled for new containers, but already running containers continue to run. Nodes in *Drain* status cannot be scheduled, and any containers running on them will be terminated and scheduled on another node.

Nodes can have user-defined labels, which are key-value String pairs. These labels can the be used to target specific devices or groups of devices when defining placement policies for containers on the swarm. For instance, a node might have the following labels: `devicetype = Smoke Alarm`, `devicefamily = RaspberryPI`. Labels are set by issuing an instruction to any of the swarm managers, which contains the hostname of the affected node and the label key and value.

2.5.2 Swarm Management

Swarm management is realized by manager nodes. They employ a consensus algorithm for maintaining a unified system view, to agree on proposed updates to the swarm, such as node additions or removals, and for electing a new leader if necessary. Management decisions are initiated by the leader, but they are agreed by other managers before being executed. Consensus algorithms enable hosts in distributed systems to agree on a value, when each host can propose a different value [32]. In the case of Docker Swarm, the Raft Consensus Algorithm is employed.

The Raft Consensus Algorithm allows decisions to be reached even when some managers are unavailable [38]. A majority of managers, also called the quorum, is required to reach consensus. If the swarm loses the quorum of managers, it cannot perform management tasks. In that eventuality, swarm tasks on existing worker nodes continue to run. However, swarm nodes cannot be added, updated, or removed, and new or existing tasks cannot be started, stopped, moved, or updated.

Setting the number of managers in a swarm is a trade-off between performance and fault-tolerance. Neither Raft nor Docker Swarm imposes a limit. Setting additional manager nodes reduces performance since more nodes must acknowledge proposals to update the swarm state. Setting too few managers can lead to a swarm where consensus cannot be reached or no managers are available. It is advisable to maintain an odd number of manager in the swarm, as this increases the chance that quorum can still be reached in case the network becomes partitioned. According the official Docker Swarm documentation, the recommended maximum number of managers in a swarm is seven [7].

Manager nodes are meant to be a stable component of the infrastructure, and

should use a fixed IP address so they can be reached even after a system reboot. If the whole swarm restarts and every manager node subsequently gets a new IP address, there is no way for any node to contact an existing manager. Therefore the swarm is stuck while nodes try to contact one another at their old IP addresses.

2.5.3 Services

Services are created to deploy applications on a swarm. A service definition includes the name of the image to use and parameters that determine how containers are configured and distributed across the swarm. These parameters allow specifying options such as the number of replicas of the image to run on the swarm, placement policies for running containers on specific nodes, ports where the swarm makes the service available to outside world, virtual networks where containers should be attached, rolling update policies and so on. The service definition constitutes the desired application state. As nodes or containers fail, the leader of the swarm takes actions to maintain this state.

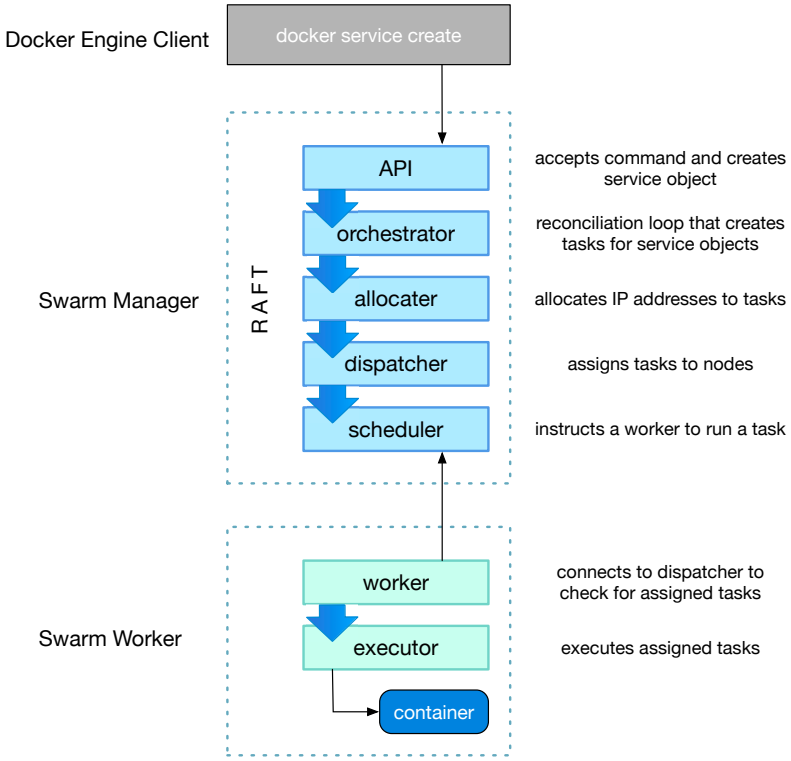

Figure 7: Lifecycle of service creation

Service definitions are sent to one of the managers by the user who operates the swarm. For this purpose, the Docker Engine command line interface or the Docker Engine API is called. The definition is forwarded to the leader of the swarm, which is the manager in charge of scheduling containers on nodes. Figure 7 shows the steps that are taken. Firstly, the orchestrator creates as many tasks as are necessary to

reach the desired state. A task is simply an object representing the container that must be created with some additional configuration. Then this task is assigned a virtual IP address, which is the IP address of the container. The tasks are assigned to nodes, according to the specified placement policies. Finally, worker nodes are instructed by the scheduler to run certain tasks. The worker receives the task and creates the container with the required configuration. Service definitions can also be updated, triggering the same cycle except that the scheduler might instruct some workers to stop running certain tasks.

There are two types of service deployment modes, **global** and **replicated**. Global services result in one task being scheduled on every available node of the cluster. That is, one container is allocated per node. Replicated services require specifying a number of desired tasks to schedule. The swarm leader distributes these tasks across the swarm, attempting to spread them over the available nodes. In this case, it is possible that some nodes may have more than one running container for the same service, while others have none. Figure 8 illustrates both types of services; the colored boxes denote containers corresponding to the service.

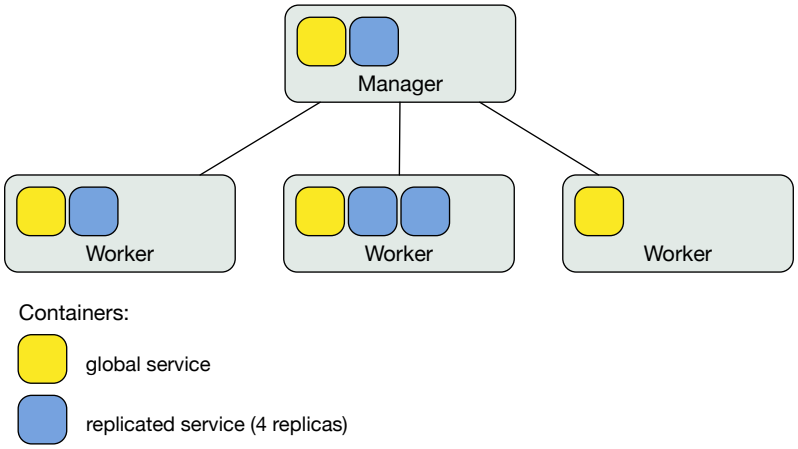

Figure 8: Global and replicated service deployment modes

Placement constraints are rules that instruct the swarm leader which nodes to consider during orchestration. These constraints are part of the service definition and are expressed as equality and inequality expressions that refer to a node label and a value. The node must satisfy these conditions to be considered by the orchestrator. Placement constraints can also be combined with deployment modes. For instance, a service can be deployed as global or replicated on the set of nodes that match the given constraints. Figure 9 illustrates several scenarios that combine placement constraints and deployment modes.

Placement preferences instruct the orchestrator to spread tasks evenly over nodes according to certain criterion. By specifying a node label name, tasks are spread out over nodes having different values of that label. More than one placement preference can be specified, in which case they are applied hierarchically. Placement preferences only make sense in the context of replicated services, and they can also be combined

with placement constraints.

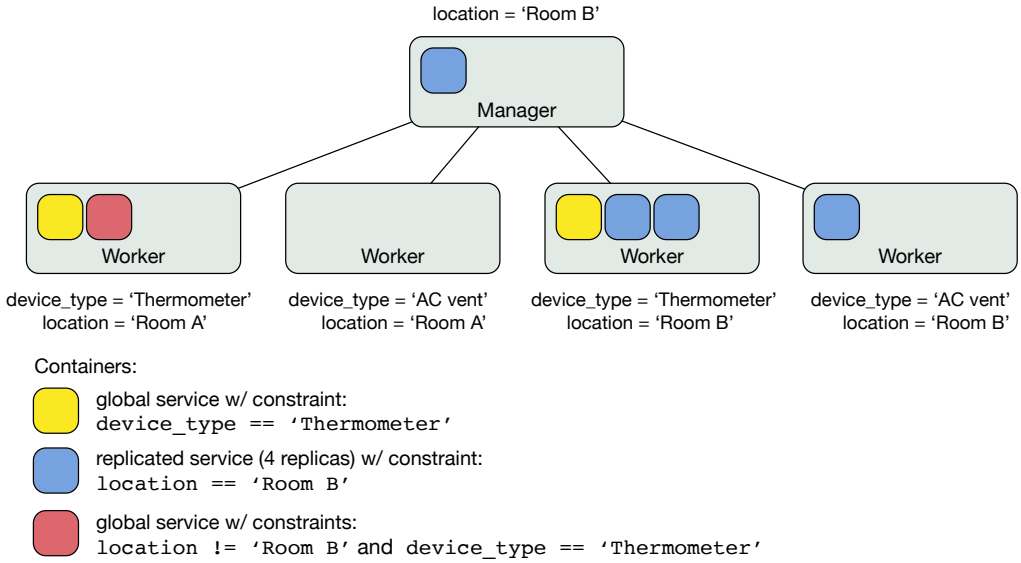

Figure 9: Placement constraints refer to node labels to deploy services on nodes that fulfill certain conditions

2.5.4 Networking

Containers running on swarm nodes interact with each other through networks. Egressing traffic from a container to the physical network is allowed by default, but incoming traffic is blocked. Enabling access, either from other services or from external hosts, requires setting up virtual networks and publishing ports.

Docker Swarm allows the creation of virtual networks where containers can communicate as if they were separate hosts on a physical network. All containers on the same Docker network have connectivity with each other on all ports. By default, all containers corresponding to the same service are connected to a virtual network and, hence, are reachable amongst themselves. However, enabling communication between containers from different services requires connecting them to the same network (Fig. 10). Network connections are defined upon service creation. A service can be a part of several networks.

Networks must be created before services can attach to them. Docker Swarm employs the *overlay* network driver for connecting containers in a swarm. Overlay networks are isolated multi-host virtual networks that abstract from the underlying physical network [5]. Containers receive a network interface and an IP address for every overlay network where they are connected. Docker Swarm manages container-to-container routing on that network.

Overlay networks are enabled by a feature of the Linux Kernel called VXLAN tunnels, which encapsulate and route container traffic in a frame of the underlying physical network. Docker Swarm creates several network interfaces to enable container-

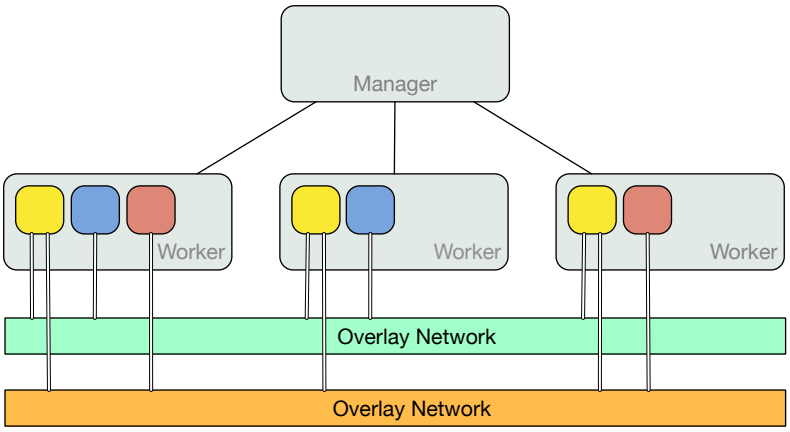

Figure 10: Services can be attached to multiple overlay networks

to-container communication. This is shown in Figure 11. For each container, a VETH pair is created, which allows traffic to flow between the container and host namespace. A bridge interface is also created and acts as a network switch. All containers on the host, which are connected to the overlay network are attached to this bridge. Finally, a VXLAN Tunnel Endpoint (VTEP) is created, which performs the encapsulation/de-capsulation and transmits packets through the underlying network. As an optimization, these interfaces are only created on hosts which are running one of the containers attached to that network.

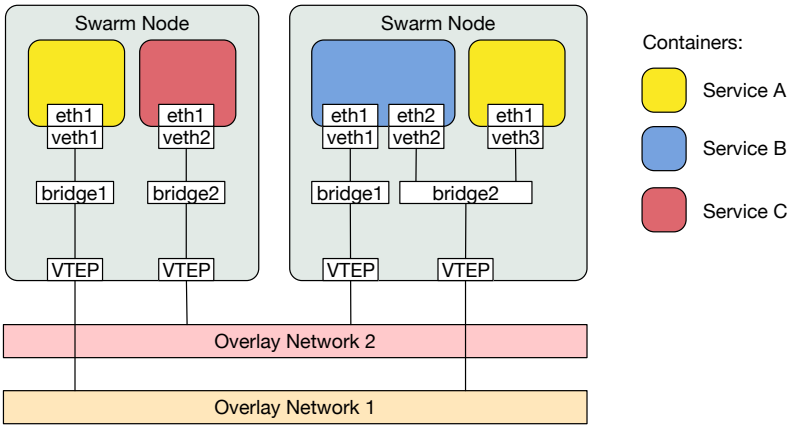

Figure 11: Implementation of overlay networks

To open a container for external access it is required to publish the ports for the service. When the service is created port mappings can be specified. Each mapping indicates an external and an internal port number. Traffic received on the external port of any host is redirected to the internal port of some container that corresponds to that service. All containers are connected to an *ingress* overlay network through which requests are delivered to the correct container (Fig. 12).

Docker Swarm implements something called *routing mesh*, which opens the external port for listening on all swarm hosts. When a request is received on that port in any host, Docker Swarm forwards the request to one of the hosts running the service.

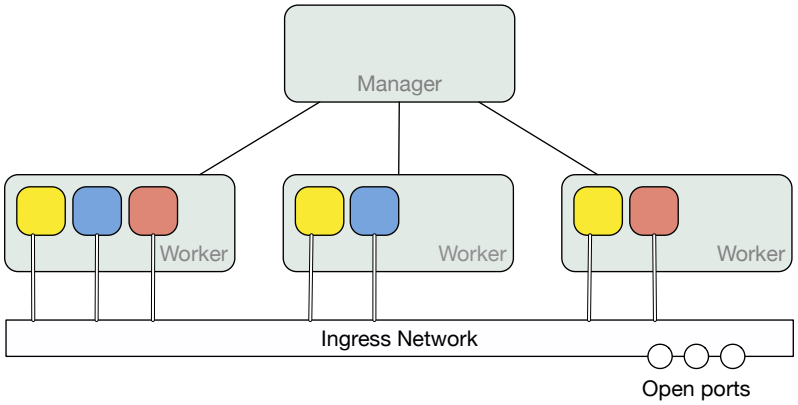

Figure 12: External access to services is achieved through exposed ports

2.5.5 Service Discovery

Service discovery refers to how requests for a given service are routed to a container of that service. Containers are dynamically allocated and IP addresses can change, making routing and discovery hard to manage manually. Docker Swarm implements mechanisms for delivering requests that originate internally to the swarm, and also those that come from external hosts, to the required container.

Applications running inside containers can call other services by service name. Service discovery is network-scoped, only containers that are on the same network can resolve each other's addresses. For two services to communicate, it is required that they are connected to the same overlay network. Each container forwards service name DNS queries to the Docker Engine, which acts as a DNS server. The Docker Engine checks if the name corresponds to a service in every network where the container is connected to. If the containers are on the same network, the DNS query is resolved. Queries that cannot be resolved are forwarded to the default DNS server.

Docker Swarm provides two implementations to resolve the query. The first one, called DNS Round Robin mode, returns the IP addresses of all containers running the service. In this case the request can be sent to any of the addresses. For the second implementation, called VIP mode, services are assigned a Virtual IP address (VIP) in every network where they are connected. The DNS query returns the VIP of the queried service. Once the VIP of the service is obtained, requests can be made directly to that address.

Requests originating externally to the swarm are handled similarly to internal requests. The main difference is that they are not called by name, but by IP address and Port number. When a service exposes a port, all hosts on the swarm listen for connections on that port. This holds even in hosts where containers of that service

are not running. This mechanism is called *routing mesh* and ensures that the request is forwarded to an appropriate container.

All services with published ports are connected to the *ingress* overlay network. When an external request arrives to any host on a published port, Docker Engine delivers it to a system container, *ingress-sbox*, which is deployed on every host and is also connected to the ingress overlay network. The ingress-sbox container performs a lookup via a port-mapping table to determine the service the request is destined to. At that point, the request is handled exactly like requests originating internally.

2.5.6 Load Balancing

Docker Swarm provides mechanisms to load-balance requests that generate internally, from a running container, and externally, from an external host. The mechanism varies if the service is using DNS Round Robin or VIP mode.

A DNS query for a service using DNS Round Robin results in a list of IP addresses of all containers corresponding to the service. The application calling the service must chose one of these addresses to make the request. This is called client-side load-balancing. This approach has two problems. Firstly, if the client DNS library implements a cache, it can become inaccurate as containers fail and new ones, with different IPs, are created. Secondly, it does not guarantee that requests are uniformly load-balanced. Actually, many DNS client libraries implement longest-prefix match, instead of random selection [23].

The preferred approach uses VIP mode. All swarm hosts employ IPVS (IP Virtual Server), a load-balancer built into the kernel [8], which directs requests to one of the hosts running a container of that service. The routing table is kept up to date, as hosts exchange control information through a Gossip network. The VIP of the service remains always the same, even as containers are orchestrated. This implementation abstracts the client from knowing the actual IPs of the containers and guarantees a more balanced distribution of requests among them.

3 From the Internet of Things to Fog Computing

The Internet of Things represents a vision in which the Internet extends to the physical world [34]. Everyday objects and machines are able to communicate, compute and coordinate between themselves [36]. Unlike traditional networked entities (e.g., routers, switches) these *things* are able to sense the physical environment (e.g., temperature, electromagnetic radiation level, movement) or to trigger actions which have an effect on the physical world, such as altering heating and ventilation in a room, controlling pressure valves in production lines, or changing traffic light timing to reduce congestion.

The steady advances in embedded systems are making this vision a reality. Processors, wireless communication modules and other electronic components are being increasingly integrated into everyday objects due to their diminishing size, constantly falling price and declining energy consumption [30]. This results in the generation of enormous amounts of data which have to be stored and processed in a seamless and efficient manner [29]. Moreover, the processing capabilities of IoT devices has reached levels where they are now capable of running computationally intensive jobs. One example is the Raspberry Pi, a popular IoT prototyping board with over 10 million units sold, which features a GPU capable of 24 GFLOPs, and 1GB of RAM[3].

It is estimated that a new era of ubiquity of IoT devices is coming, where humans may become the minority as generators and receivers of traffic [20]. In such a perspective, the conventional concept of the Internet as a network infrastructure connecting end-user's terminals will fade, leaving space to a notion of interconnected "smart" objects forming pervasive computing environments [36]. However, the Internet will not disappear, but rather turn into a global backbone for worldwide information sharing and diffusion, interconnecting physical objects with computing-communication capabilities across a wide range of services and technologies.

In fact, the stack of protocols that make up the Internet already connect a huge amount of communicating devices and run on battery operated embedded devices [15]. With some adaptations and simplifications it is suitable for making IoT a reality. Moreover, existing routing infrastructure could be used. But IP is not the only one of Internet technologies on which IoT will be based. Some authors have proposed using HTTP and REST [28] as the basis for a Web of Things architecture, integrating smart things not only to the Internet (i.e., to the network), but also to the Web (i.e., to the application layer). This API-oriented approach enables heterogeneous devices to "speak" the same language, and would allow building new applications by recombining existing ones [30].

Still, the heterogeneity of the IoT presents additional challenges beyond using a common interaction language. In fact, IoT devices are heterogeneous in many aspects. They have different hardware characteristics (e.g., processor, memory, sensors, actuators), are deployed under different conditions (e.g., mobile, static) and use different networking technologies (e.g., WiFi, LTE, LoRa), they are built by different manufacturers and serve different purposes. Moreover, some are always on,

[3] https://www.raspberrypi.org/help/faqs/#performanceSpeed

while others are intermittently active to preserve battery. The applications running on them are also diverse and dynamic, with different application-specific requirements in terms of latency, bandwidth and security. All of this has led to the development of new computing models, which harness the computing power of these devices while overcoming the limitations of traditional centralized computing models.

3.1 Device Heterogeneity

The IoT is made up of highly heterogeneous and resource-constrained embedded devices. Due to incompatible architectures, different operating systems, missing libraries, among others, programs have to be developed or fine-tuned for each supported device type. Deployment and orchestration of applications is also a hard task, as each of these types has its own way of being updated and configured. To successfully develop and deploy IoT applications, these devices should share a general software infrastructure that abstracts the heterogeneity, while adding minimal overhead, and allowing for efficient management and operation [22].

Several software abstractions have been developed to provide a unified development environment for distributed applications involving embedded devices. One approach is to develop operating systems that decouple the software programs running on them from the actual hardware through and abstraction layer and platform independent APIs [33]. However, these systems require using their own programming model, which excludes the possibility of using already existing programs without rewriting them.

Another approach is to use stripped-down versions of Linux, specifically designed for certain platforms [24]. However, this is more often than not targeted at devices with very low-computational power. By stripping down features to port Linux to these devices, the result are Linux-based yet POSIX non-compliant operating systems. This makes integration of new software a process that has to be done case-by-case for each particular operating system. These standards are in place to provide compatibility, and modifying them can lead to unexpected behavior and bugs.

Virtual machines (VMs) could be used to solve these problems, by compiling applications into a machine-independent form (referred to as *bytecode*) that is then run by an interpreter. The same application could then be run in any hardware platform with a suitable interpreter. The Android operating system is one such example. Its core is based on the Linux kernel, while applications are written in Java. Originally, Android was conceived as an Operating System for mobile phones, which would tackle device heterogeneity and enable developers to have their applications run on any device by providing a set of APIs which abstract hardware differences. More recently, Android Things has been announced, which is aiming at doing the same thing for embedded systems. Although a Java-compliant VM is expressive to encompass different application scenarios, it still incurs in overhead that is non-negligible. In the case of Android, these platforms are closely tied to Google services for provisioning or updating apps, which in some scenarios might be undesirable.

In the cloud computing context, lightweight virtualization based on software

containers has recently gained momentum [45], as it allows to easily create, deploy and scale distributed applications over multiple data centers. By using features built-in recent versions of the Linux kernel, these solutions allow the creation of virtual environments, without the overhead of virtual machines and hypervisors. Applications are packaged along with all their dependencies into images that can be instantiated in any Linux environment that supports. Several software platforms have emerged that orchestrate the deployment of these images across a pool of connected computers. In the field of IoT, these solutions have only recently began to be explored, but with promising results, as they overcome platform heterogeneity, while inducing low overhead. In the following chapter, this topic is explored in depth.

3.2 Computational Models

Cloud Computing is a model that enables ubiquitous, on-demand access to shared computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [35]. This comes at an affordable price, mainly due to the economies of scale: higher predictability through massive aggregation; carefully selected locations with inexpensive power; and lower operational costs achieved through the deployment of homogeneous compute, storage, and networking components [19]. For these reasons, Cloud Computing has emerged as the main computing model powering most of today's web and mobile applications. End-devices (e.g., laptops, tablets, phones) offload most computations and storage to centralized servers. This is specially convenient for users, allowing access to the same service from any device, with little distinction if using a fast computer or a somewhat less-powerful phone.

However, in the coming years, the demands for compute and storage resources will come mainly from IoT devices. Billions of fixed and mobile endpoints will span vast geographical areas and will be arranged in various different forms, covering a myriad of different use cases [48]. Many of these settings will have stringent requirements, such as very low latency, and fast decision making based on real-time analytics. Some deployments might be made in places where the communications with the cloud are either too expensive or unreliable (e.g., due to poor connectivity). In many of these scenarios, cloud computing is simply not an option. In others, it makes sense to take advantage of under-used locally available resources. In fact, the capabilities of end-user and edge devices have become suitable for running computationally intensive jobs and storing large data.

Fog networking is an extension of the cloud-based networking model that addresses these challenges [21]. The name Fog comes from the analogy of bringing Clouds closer to the ground (i.e., edge and end-devices). In particular, multiple edge and end-user devices form their own cloud, and collaboratively carry out a substantial amount of computation, storage, communication and management, rather than deferring it to a centralized cloud. In this way, the aforementioned benefits can be achieved, namely: real-time processing for applications with strict delay requirements that cannot afford communication with the cloud; meeting client-centric benefits (e.g., geographical, performance, privacy) by using near-by devices; using a pool of often

under-utilized local resources.

It is worth noting that Fog is envisioned as an ally, and not a replacement, of Cloud Computing for a broad spectrum of IoT scenarios. In many cases both models will be combined to reach client objectives [48]. Some computations, communications and storage can be carried out in the IoT devices, while the cloud retains it's role in coordinating different fogs, and in aggregating data from different locations for management purposes (e.g., business intelligence).

3.3 Software Containers in Fog Computing

The performance overhead of software containers and Docker in particular has been evaluated in various works. In cloud environments it has already been shown that containers provide significantly less overhead that traditional virtual machines. However, they do incur in additional overhead with respect to applications running directly on the host system particularly in network and I/O operations [27]. In IoT environments Docker has also been evaluated. The authors in [43] benchmark CPU, memory, disk I/O and Network I/O performance on an resource-constrained Cutieboard2 comparing native, Docker-based and KVM virtualization execution. The results obtained indicate that containers offer better performance than virtual machines, and near native performance in some cases. The authors in [37] run performance tests on two models of Raspberry Pis and show that there is little overhead when running containerized applications over host applications in terms of execution time, memory and disk access speeds, and power consumption.

There have already been some container-based approaches for managing and operating Internet of Things devices. In [31], the author proposes using Docker containers on a set of Raspberry Pis managed through Kubernetes. A camera-based security/surveillance application is built where the devices pre-process the captured frames, then selectively decide to forward camera frames to the cloud based on detecting movements or not. The solution handles disconnection of nodes from the Internet without data loss by using neighboring devices to provide replication. However, one of the setbacks of the implementation is that it requires manual setup of the cluster, i.e., specifying the IP of each node and each master and executing join commands in each of them. While this is possible for a small collection of devices, it is impractical in the IoT where millions of devices are connected. Furthermore, while the heterogeneity of devices is considered at an application level, with different devices running different applications, the heterogeneity at a hardware level is not considered.

Authors in [17] employ Docker Swarm to manage multiple IoT gateways. They extend an existing open-source framework for building IoT gateways (Kura) to support running Docker Engine. The gateways are provisioned with some basic configuration, and each service or task to be executed is specified as a different container. This allows installing, replacing, or extending available middleware services without affecting other running services. By forming a cluster of gateways they are able to provide a fallback for failing gateways, load-balancing, and to have specialized gateways (e.g., logging, analytics, processing, inputs, outputs). The management of this cluster

happens at the cloud layer, allowing for a global overview and management of the system status. The authors select Docker Swarm over other Mesos and Kubernetes due to it being more lightweight than existing alternatives and for the active developer community and growing industry support.

The authors of LEONORE propose a scalable framework for provisioning large-scale IoT deployments composed of resource-constrained devices [47]. They also consider that applications change overtime, which makes update mechanisms necessary, and should be performed fast and efficiently. For this, they design an architecture through which minimal application packages containing applications and their dependencies can be deployed on devices. These applications are then run inside a runtime container which provides isolation from the host system. One important aspect of their design is that simplifies large-scale deployments by bootstrapping device registration. Upon startup, each managed device registers with the framework by providing its unique identifier, which is derived from a combination of parameters, including MAC address. Once the registration is finished, the device can be remotely provisioned by the framework. Although their framework is flexible and scalable, it requires using custom LEONORE tools and application packages, which are not as widespread and supported as other solutions (i.e., Docker). Furthermore, the solution does not contemplate composition of the applications running, for instance, by means of virtual networks.

3.4 Resin.io

Resin.io is a platform which simplifies the deployment, update and maintenance of code running on remote IoT devices[4]. IoT devices are typically hard to update and many times require physical access. Resin.io enables using development and deployment workflows and tools for provisioning IoT devices that are similar to those used for cloud-based web applications. In this way, frequent incremental updates can be easily achieved with minimal downtime.

The platform provides features such as device monitoring, log collection and shell access to devices. However, the key distinctive characteristic is that applications are deployed as Docker containers instead of running directly on the host operating system. This approach has several benefits. First, applications are packaged in images, containing all necessary software dependencies and can be executed without modifying the host operating system. Second, Docker Engine already provides mechanisms for fetching and running remote images. This frees developers from having to implement their own mechanisms to replace and re-launch running applications. Finally, Docker images are read-only, greatly reducing the risk of an attack modifying the running code.

Devices connected to the Resin.io platform are required to run ResinOS, a Linux-based operating system, which includes Docker Engine and software tools which enable remote management. Operating system images are provided for several popular embedded systems for the IoT, such as Raspberry Pi and Intel Edison. There

[4] https://docs.resin.io/introduction/

are also instructions on how to add support for other systems. The image includes a management container which is always running and interfaces with the Resin.io service (i.e., logging, monitoring, receiving commands). All operating system files are mounted read-only, which guarantees that even if an application container is compromised, the whole system remains safe.

The operating system provided by Resin.io can be updated remotely as well. A seamless update system is employed that relies on two partitions on the system storage, one stores the running operating system and the other is used for downloading an updated version. Once the download is complete, the device is rebooted by the management container and the device is booted from the partition that was previously inactive. This reduces update time and eliminates the risk of a partial update, as there is always at least one working boot partition.

To connect a device to the platform, the developer downloads the ResinOS image and flashes each board manually. For each device, a configuration file must be generated through the Resin.io web platform and placed on the image before flashing. When a device first boots up, it registers itself with platform and can be managed through an online dashboard thereafter.

Devices of the same type (e.g., Raspberry Pi) can be grouped into what Resin.io calls *applications*. The devices conforming an application all execute the same code and provisioned with a single instruction. This is achieved through a version control repository hosted by Resin.io. Whenever code updates are pushed to the repository (component 1 in Figure 13), a process is triggered which builds and updates the container images running on the devices.

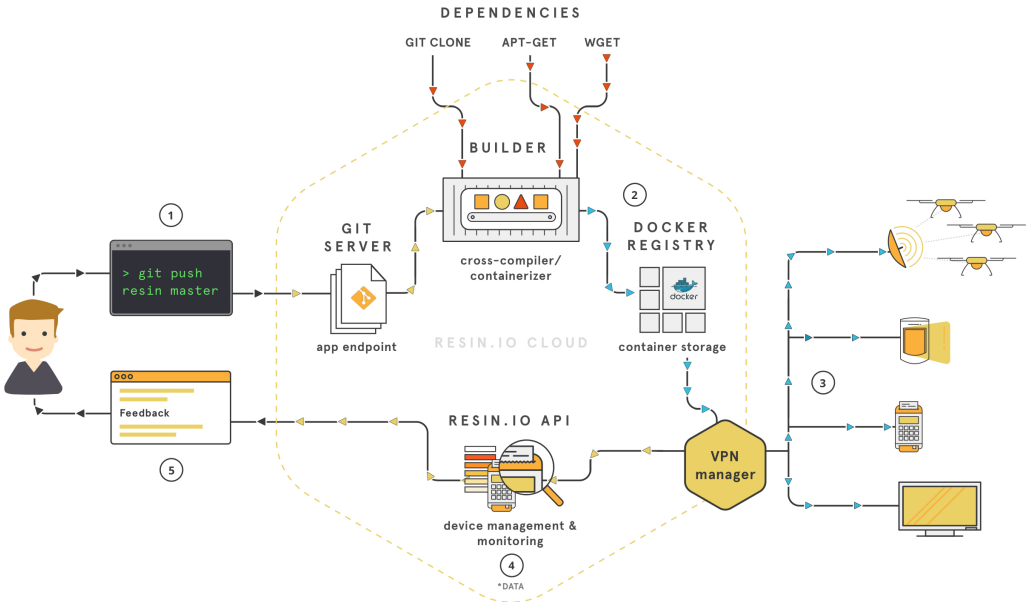

Figure 13: Resin.io architecture

Most of the work is done on the server side. Resin.io[5] builds the Docker image for

[5] https://docs.resin.io/understanding/understanding-code-deployment/

the architecture of the target devices and stores it in a private registry (component 2 in Figure 13). Then the server instructs devices to fetch the new image by communicating with the management container which runs on each device (component 3 in Figure 13). Since Docker images are based on layers, only the modified layers need to be downloaded at each device. Therefore, small changes do not lead to a whole image being sent. Once the image is available on the device, the management container executes a container based on it. Only when the new container is running, the previous one is stopped. This approach minimizes the risk of replacing a working code version with a faulty one. The web platform collects device and application data and provides remote device management and monitoring (components 4 and 5 in Figure 13).

Resin.io greatly simplifies provisioning of IoT devices by clever employment of Docker containers. However, several other advantages of Docker are disregarded. For instance, the platform limits the execution to one container per device. In that sense, the parallel nature of Docker is not used. Also, applications cannot be deployed across heterogeneous device types (e.g., all must be Raspberry Pi). In that regard, the solution does not leverage on the portability of Docker containers. Finally, the solution is focused on stand-alone devices on on devices that only interact with the cloud. Resin.io provides no mechanisms for enabling inter-device communication, composing applications, using locally-neighboring devices as fallbacks to prevent data loss or for performing distributed computing tasks.

4 Design and Implementation

IoT systems consist of resource-constrained heterogeneous devices that are deployed in diverse environments. Deployments encompass hundreds of devices that are placed in hard-to-reach remote locations. In the context of fog computing, these devices run distributed applications that interact among themselves and with external applications. This chapter describes the design and implementation of a distributed computing framework for IoT devices based on software containers.

First the system architecture is presented and described. This is followed by an explanation of how the four target properties of the system were met, namely: remote management and provisioning of devices; simple automated set-up; support for resource-constrained heterogeneous devices; and deployment of distributed applications.

4.1 System Architecture

An architecture that combines cloud and fog computing was designed. Edge and end-devices conform multiple fogs by means of Docker Swarm. Docker Swarm is chosen due to the ease of deployment, given that it is a part of Docker Engine (version 1.12 or higher), devices running Docker Engine can participate in a swarm without additional software and with minimal configuration. Moreover, many of the existing tools for managing a stand-alone Docker Engine can be used for managing a swarm and the syntax for deploying a container on a single device is very similar to that for deploying a service.

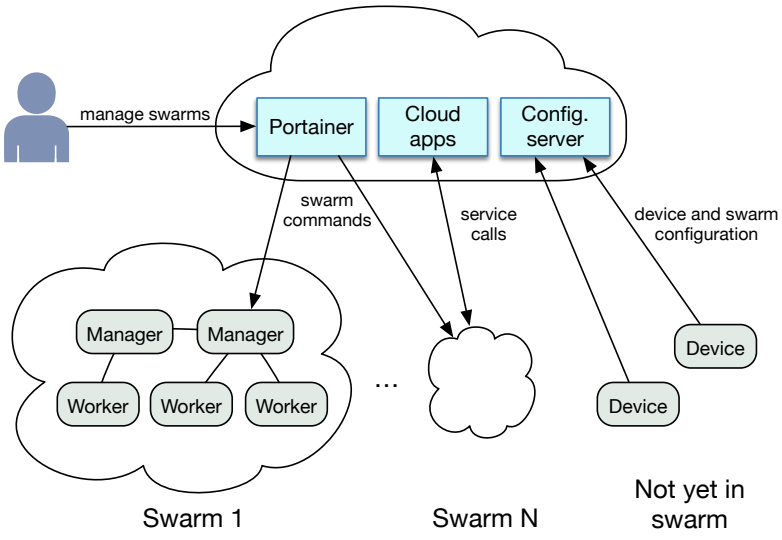

Figure 14: System architecture

Distributed applications are deployed and orchestrated on these swarms. A cloud-based management UI enables remote management of the swarms and deploying applications on them. There is a configuration service in the cloud that holds device

and swarm configuration, which is used to automate the initialization of the cluster. Figure 14 illustrates the architecture.

In each swarm there are nodes with manager and worker roles. The architecture places no restriction on the size and span of a swarm, nor on the choice of swarm managers. However, a swarm should be constructed so that communications between the nodes is reliable and latency is low. Unreliable communication between managers can lead to the swarm partitioning into more sets, where services cannot be deployed across multiple sets. Managers maintain swarm state information through the Raft consensus algorithm, which requires the exchange control messages between them. If consensus cannot be reached (i.e., because there is no quorum), then the swarm will fail to accept management tasks. It is possible to create a swarm with a single manager, however, if it fails, the swarm becomes unrecoverable. Hence, other nodes should be setup as managers as well.

Selection of the leader, managers and workers is left to the user. Gateways are commonly used in IoT deployments as devices with enhanced hardware capabilities and reliable connection to the Internet and to end-devices. In such cases, the gateway provides a good choice for leader of the swarm. All the devices that communicate through that gateway belong to the same swarm. If it makes sense to orchestrate services on devices served by different gateways, then a larger swarm can be formed, where one gateway acts as leader and the others act as managers. However, due to restrictions of the Docker Swarm implementation, individual nodes can only participate in a single swarm.

4.2 Remote Management

Management of the swarm is achieved through Portainer, an open-source web application that enables the management of Docker Engines through a visual interface. The main advantages of Portainer are the ease of installation with minimal configuration and that it provides broad control over Docker Engine functionality, including swarm management through a simple UI. Among its strengths are allowing management of multiple swarms and providing an access rights system for multiple users. For instance, it is possible to grant a user management access to a subset of managed swarms, or services running on those swarms.

Through Portainer services can be remotely deployed and managed. When a service is deployed through Portainer, all parameters can be specified, such as image, scheduling and placement policy, port mappings and networks, update policy, and access to host volumes. Through this configuration it is possible to specify which devices should run containers for a given service as well as how multiple services can communicate between them or with external programs.

Nodes connected to Portainer are referred to as endpoints. Each of the endpoints can be managed through Portainer. However, if the node is the manager of a swarm, it is also possible to deploy and manage services on the whole swarm . To enable this connection between Portainer and a manager, it is necessary to provide remote network-access to the Docker Engine running on the manager (Fig. 15). For this purpose a thin software layer was developed that modifies the way in which the

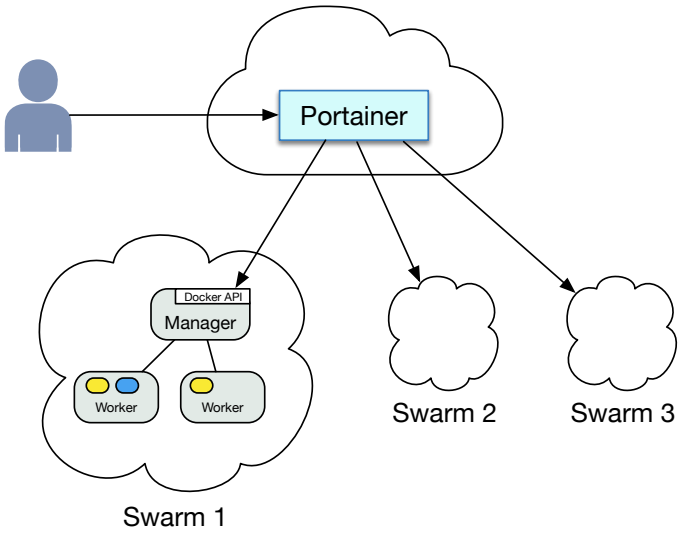

Figure 15: Portainer enables remote swarm management through a connection to the swarm manager

Docker Engine is initialized. Normally, the Docker Engine daemon creates a UNIX socket where it receives API commands from the client. By passing modifying the initialization parameters of the daemon, it is possible to use a TCP socket as well, which is published on a specified port. After this, external hosts, including Portainer, can manage the local Docker Engine. Docker enforces security to this socket through TLS certificates, which provide proof of identity and encryption[6].

4.3 Swarm Initialization

Docker Swarm requires manual intervention from the user for the initial setup phase. The user executes a command on the leader to create a swarm, and then commands on workers for joining. This involves connecting remotely to each device, executing a set of commands with parameters such as the IP of other devices, and transferring security tokens from device to device. This approach is impractical and time-consuming for the IoT, which is characterized by a vast scale of devices. To overcome such limitation, a configuration server is designed, which holds all the relevant information for cluster initialization. Devices retrieve their configuration upon startup and are able to form the cluster without any manual intervention.

Upon their first initialization, devices lack information about the swarm they belong to and their node labels. At startup, a container is run that fetches a configuration file from the cloud server (Fig. 16). This container is run with access to the local Docker Engine socket, so that it can call Docker commands. The device passes its unique identifier (i.e., MAC address) to the configuration server, which uses it to find and return the corresponding configuration file. This file contains

[6] https://docs.docker.com/engine/security/https/

information about node role (e.g., worker, manager), how to reach the swarm leader (e.g., at which IP and port), security tokens required to join the swarm, as well as node labels which describe the device (e.g., sensors, actuators, location), as can be seen in listings 6 and 7.

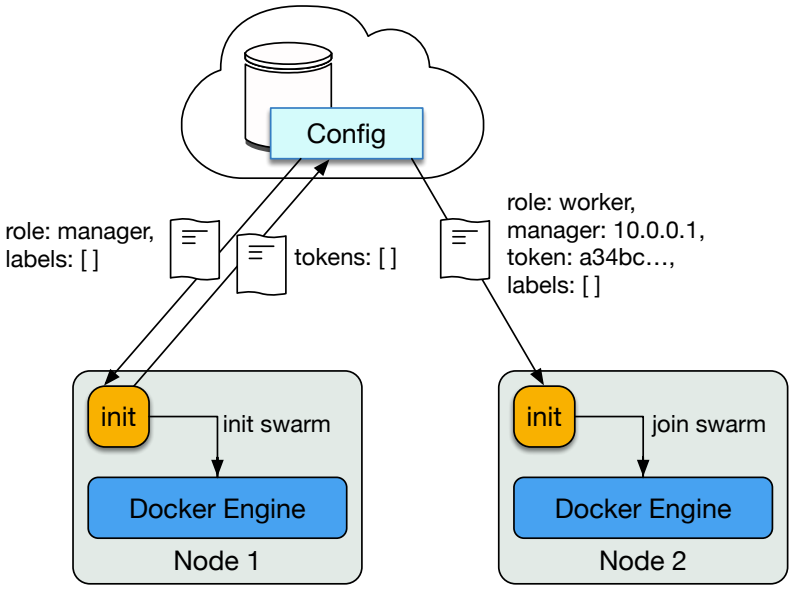

Figure 16: Devices obtain initial swarm configuration from a configuration server in the cloud

After the configuration file is retrieved, the initialization container issues commands to the local Docker Engine. In the case of the leader of the swarm, the container creates the swarm and provides the generated security tokens back to the configuration server. In the case of other manager or worker nodes, they join the swarm by passing the address of the leader and the security token. It is possible tokens are not available at the time that devices consult the configuration server. In this case, the operation is retried periodically until successful.

Devices join the swarm with a *drain* status. This means that they are part of the cluster, yet no services can be scheduled on them. Before they can be scheduled, they must inform the leader of their node labels, present in the configuration file. Labels are updated by calling the swarm leader's Docker API from each node. After this happens, nodes become schedulable by updating their availability to *active* through another API call.

The swarm managers must also register themselves as endpoints in Portainer, so that they can be managed remotely. Once the leader has created the swarm or the other managers have joined, they each call the Portainer API. Once they are registered, distributed applications can be remotely deployed and managed on the whole swarm.

Through the use of a configuration server and initialization containers the set-up of a swarm is automated. Devices can be provisioned with the same generic OS image without any device-specific configuration. This greatly facilitates and reduces

time needed for the deployment large quantities of devices. Moreover, since device- and swarm-specific configuration is stored in the cloud it is possible to update after initial device deployment, without having to connect to each device one by one.

```
{
   "role" : "worker",
   "swarm" : {
      "leader" : "10.0.0.1:2377",
      "managers": ["10.0.0.2:2377","10.0.0.3:2377"],
      "token": "SWMTKN-1-5hnmaudycoetnkik8c4dj0yyidvsdrvaxa44krpa7fdwibk0tu-705
          ygjowshn66ji5wqsctr9yo"
   },
   "labels": [
      {
         "key": "humidity_sensor",
         "value": "1"
      },
      {
         "key": "humidity_sensor_type",
         "value": "AM2302"
      },
      {
         "key": "location",
         "value": "Room A134"
      },
   ]
}
```

Listing 6: Sample configuration file for a worker node

```
{
   "role" : "leader",
   "labels": [
      {
         "key": "humidity_sensor",
         "value": "1"
      },
      {
         "key": "humidity_sensor_type",
         "value": "AM2302"
      },
      {
         "key": "location",
         "value": "Room A134"
      },
   ]
}
```

Listing 7: Example configuration file for the swarm leader

4.4 Device Heterogeneity

The heterogeneity of IoT devices presents itself in the form of varying microprocessor architecture (e.g., amd64, ARM, i386), system resources (e.g., CPU, memory, storage) and availability of sensors and actuators. In many cases, it is required to abstract the heterogeneity and build applications that can run on any device, for instance,

running a routine that monitors uptime and processor load on every device. In other cases, it is required to exploit the heterogeneity and deploy applications only on devices that fulfill certain criteria, for instance, deploying a software update only on devices of a specific type, make and model. Some scenarios require abstracting certain characteristics while exposing others, for instance, running a temperature collection application on all devices with a thermometer, regardless of the underlying hardware characteristics.

Abstraction is achieved by building images compatible with multiple processor architectures. Discrimination is achieved through device metadata, called labels, that describe their capabilities. These techniques are detailed next.

4.4.1 Multi-Architecture Images

Docker images include files, libraries, and binaries compiled for a given hardware architecture; consequently, they can only run on that specific architecture. To deploy applications on heterogeneous devices, developers need know beforehand which architectures they will target, build the images accordingly and make them available to the devices, typically by uploading them to some registry. When deploying services on a swarm of heterogeneous devices, there is no way to instruct Docker to use a different image for each architecture. If a mismatching image is scheduled on a node, Docker issues no warning about it, but executing the container fails and Docker enters a loop of continuous restart attempts.

The solution employed in this research is a feature of the Docker Registry Image Manifest, particularly Version 2 Schema 2, which adds support to multi-architecture images, or fat manifests[7]. Fat manifests allow a registry to hold multiple images for different architectures under the same name and version tag (Fig. 17). When a device performs a query for a given image name and version, the registry returns a manifest listing of all available architectures and the corresponding image file (i.e., via an identifying digest). The device then downloads the image matching its architecture. Listing 5 shows a sample manifest which supports two architectures. In the absence of a match, one is randomly selected, leading to an initialization error as previously described. However, as long as the developer includes all the necessary images in the registry, this approach works seamlessly with swarm services, thus enabling the deployment of multi-architecture services.

Although fat manifests are part of the Docker Registry specification, there is no official tool to generate multi-arch images. For this reason, an external open-source *manifest-tool* [9] was employed to consolidate existing images into a multi-architecture image.

The process to build multi-architecture images is depicted in Figure 18. A Dockerfile must be provided by the developer for each supported architecture (step 1). Then the corresponding images must be built and pushed to a registry (steps 2 and 3). At this point, different image names or version tags are used (e.g., temp-reader:arm, temp-reader:amd64). When all images are uploaded, the manifest-tool is called, which consolidates the multiple images into a single multi-arch image (step 4).

[7] https://docs.docker.com/registry/spec/manifest-v2-2/

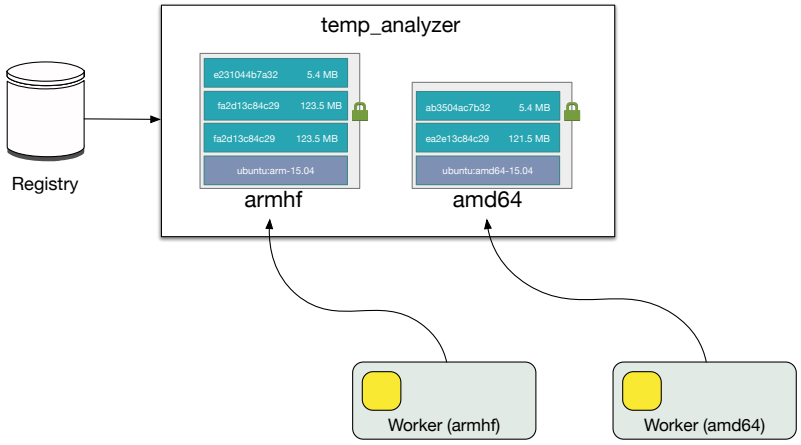

Figure 17: Multiple architecture images are listed in a single manifest. Each device retrieves the one matching its architecture

Producing multi-architecture images is required for deploying applications on heterogeneous devices, but is not sufficient. These devices are also resource-constrained, hence the produced images must be as small as possible. In some cases devices have limited storage, while in others bandwidth is costly or restricted. By reducing image size, deployment of new applications is also realized faster.

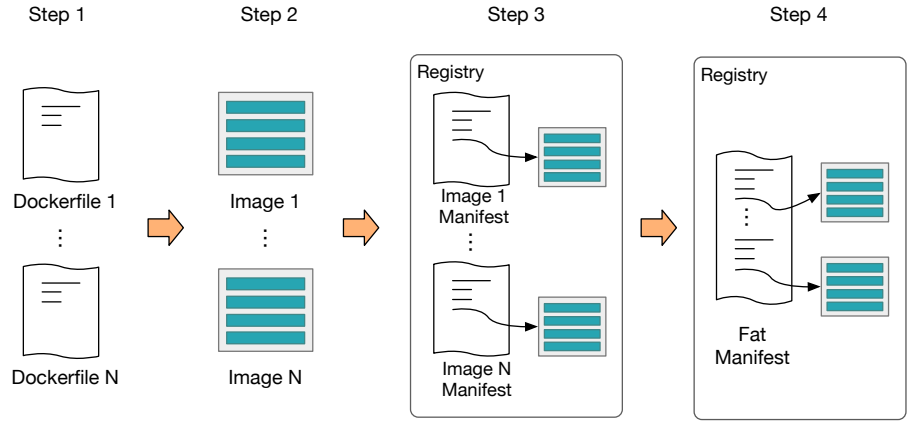

Figure 18: Multi-arch images are built with a series of steps: single architecture images must be built first and pushed to a registry before consolidation

Multi-stage builds are employed to achieve image size reduction. The first stages of a Dockerfile download dependencies and prepare the application. Only those programs and libraries required at runtime are copied into the last stage, which conforms the final image. In this way, all of the overhead of compilers, package managers and other tools which are present in typical container images is avoided. This technique is also applied to build images for architectures different from that of the computer where they are built. The first build layers are based on architecture

of the developer's host (e.g., x86) and cross-compile the application for the desired architecture. The last layer is based on an image capable of running on the target architecture, and the artifacts produced by the previous layers are copied onto it.

4.4.2 Node Labels

Devices realizing IoT deployments have diverse capabilities and are deployed under varying conditions. The capabilities vary according to presence of certain types of sensors and actuators, processing power, system memory and storage, and so on. The conditions vary according to power source type (e.g., direct power, batteries, solar panels), connection type (e.g., WiFi, GSM, LoRa), location, among others. This heterogeneity is modeled to allow scheduling applications on specific groups of devices that match possess certain characteristics.

For this purpose, Docker Swarm node labels are employed. Device characteristics are expressed as key-value pairs, which denote a property and the value of that property. Table 1 shows some examples of how labels can represent device characteristics. For instance, they can be used are to indicate the presence of a certain characteristic, such as a type of sensor, with a boolean value of `1`, or also more specific information, such as a model number. These labels are then used to deploy applications on devices with specific characteristics.

Category	Example labels

Sensors and actuators	`node.labels.humidity_sensor = 1`
	`node.labels.humidity_sensor_type = AM2302`
	`node.labels.air_vent_actuator = 1`
	`node.labels.air_vent_actuator_type = FS90R`

System characteristics	`node.labels.device_make = raspberry pi`
	`node.labels.device_model = 3B`
	`node.labels.arch = armv8`
	`node.labels.cpu_cores = 4`
	`node.labels.ram_memory = 1 GB`

Location	`node.labels.location = Washington Road` `node.labels.outdoors = 1`
Connection	`node.labels.connection_type = wifi` `node.labels.connection_reliability = high`
Power source	`node.labels.power_source = power line` `node.labels.power_reliability = high`
Security	`node.labels.performance_critical = 1`

Table 1: Sample node labels in different contexts

Placement constraints are specified when deploying or updating services on the swarm. They reference node labels as conditions that target devices where the service can be executed. Containers are orchestrated only on those devices that match the specified constraints. In this way, it is possible to target devices by a

given characteristic. For instance, it is possible to deploy a temperature collection service across all devices that possess a thermometer, regardless of their device type, architecture, function or other features. It is also possible to combine placement constraint with other features of services placement, such as global and replicated modes.

4.5 Distributed Applications

The previous sections describe the steps needed to conform and manage clusters of heterogeneous devices. These efforts lead to the goal of running distributed applications on the devices. Deployed applications must be able to interact with each other with minimal latency; they must also be able to interact with external systems such as those hosted in the cloud; they must be able to adapt to changes in the cluster, such as those caused by disconnecting devices; and they must be updateable in a way that guarantees uptime. Docker Swarm was the chosen framework for these purposes, since it already includes many of these features. In this section, the features of Docker Swarm that allow reaching these goals are described.

Docker Swarm allows managing a collection of heterogeneous devices as a computing cluster and deploying applications as services on it[12]. There are many features that make it a convenient choice, such as the possibility of deploying services on subsets of devices, the ease of composing deployed services, built-in load-balancing, and desired state reconciliation. Devices on a swarm can run distributed computations. The swarm status is actively monitored by the managers, which orchestrate containers to maintain desired service status. If a node or a container fails, another node can pick up the work load. Moreover, services running on the swarm can call one another without a cloud middleware, leading to low-latency communications. Although communication with the cloud is also possible in both directions, originating from devices, or originating from the cloud.

Service Placement

When services are created, scheduling modes and placement constraints give control over which devices run the deployed application. Docker Swarm supports global and replicated scheduling modes, which determine if a container is deployed on each device in the swarm, or if a specific number of replicas is scheduled across the swarm. These modes can be combined with placement constraints to limit the set of devices where containers are scheduled.

When used in combination with user-defined node labels, such as those in Table 1, placement constraints enable fine-grained control over the placement of containers on the swarm. It is possible, for instance, to specify that a service should be deployed on all nodes that possess a given sensor, or that are located in a certain space.

Service Composition

Containers running on the swarm can communicate with each other through overlay networks. When services are deployed it is possible to specify multiple networks where

containers of that service are attached. Communication between those containers happens just as hosts communicate on a local network. Calls are made by service name, which Docker Swarm resolves into the IP address of individual containers. If nodes become unavailable or unreachable, services continue to be accessible without changing any configuration. Whereas services connected to the same network can be composed, services which are not connected to a common network remain completely isolated, thus, providing additional security.

External Access

In many scenarios, communication with external services or hosts is required. For instance, sensor data can be processed in the fog and be sent off to remote servers for business intelligence or permanent storage. Communication with external services can originate from the swarm or from external hosts.

Outgoing connections from the swarm to external hosts are enabled by default. Containers deployed on the swarm can call any cloud-based services, as long as the network configurations (e.g., firewall rules) allow. A scenario where this type of connection is employed is when devices push their data to external hosts for back-up.

Incoming connections to running services are also possible, but must be explicitly allowed. When a service is deployed on the swarm, ports are exposed. The indicated port is open for listening on every node of the swarm. When an incoming request arrives, the swarm delivers it to one of the running containers of that service. Requests are load-balances across the different containers. This type of connection is useful for scenarios where remote servers poll devices periodically for data.

Rolling Updates

One common limitation of IoT deployments is provisioning devices with new versions of their running applications. Devices are typically configured in the factory and never updated again. Docker swarm allows updating running services by specifying a new image for the service. The update is performed in a rolling manner, where containers are replaced sequentially, providing updates without any downtime. It is possible to specify the update parallelism, which is the maximum number of tasks that can be updated simultaneously, and time between each container instantiation.

5 Results

The proposed solution was evaluated through qualitative and quantitative methods. This chapter presents the results of the evaluation and is divided in two parts. The first part provides a thorough comparison with an existing service for IoT provisioning through software containers is made. The second part compares container images produced by using standard techniques as well as those described in the previous chapter to evaluate the reduction in image size under different scenarios.

5.1 Comparative Analysis

The previous chapter presented a framework for distributed IoT applications based on software containers. To evaluate the proposed solution, six target properties stemming from the characteristics and requirements of IoT deployments are defined. IoT deployments are composed of vast amounts of heterogeneous devices installed in varying locations. Moreover, applications running on these devices interact on the edge of the network and are frequently updated as user requirements change. Considering this, the target properties are the following:

- **Simple set up.** The initial installation of operating system and required tools for the framework should be simple and fast. Manual configuration should be minimized.
- **Remote management.** IoT devices should be remotely manageable. Applications should be deployed from a single control interface that transmits the instructions to each device. Operations should not require connecting to each device and manually executing commands.
- **Device heterogeneity.** Devices with different hardware characteristics – such as processor architecture, system resources, and available sensors and actuators – should be supported.
- **Application heterogeneity.** The same application can be executed on different types of devices, abstracting the underlying hardware heterogeneity.
- **Multiple applications per device.** Multiple applications should be executed in parallel on each device. Different tasks, such as data collection, data processing, storage, and so forth run inside isolated containers. If one task fails, it does not compromise the entire system.
- **Application composition.** Applications running on end-devices should be able to interact through APIs exposed to other applications. The actual devices where an application runs should be transparently accessed.

A comparison with Resin.io (Section 3.4) is made on these six points. Resin.io is chosen as it is the main commercial provider of an IoT provisioning solution based on software containers. It is a solution which cleverly employs Docker tools to simplify

software updates on remote devices. However, it takes little consideration of other aspects of IoT applications, such as interactions between applications and device heterogeneity. The results of the comparison are summarized in Table 2 and detailed next.

Feature	Resin.io	This Work
Simple set up	No	Yes
Remote management	Yes	Yes
Device heterogeneity	Yes	Yes
Application heterogeneity	No	Yes
Multitasking	No	Yes
Applications composition	No	Yes

Table 2: Comparative analysis between Resin.io and this work

Simple Set Up

This characteristic considers how simple it is to install and configure the required tools on IoT devices, so that devices can be managed and applications executed. In the case of Resin.io, this involves installing a custom operating system image and registering the devices on an online platform. In contrast, the proposed solution requires installing an operating system with Docker Engine, registering the devices on Portainer, and setting up a swarm. Although the latter comprises one extra step, the set up process is simplified when compared to Resin.io.

Resin.io requires that a custom OS is installed on each device. In fact, such an image contains programs that connect to the online platform and allow devices to be managed remotely. However, the same image can not be used for every device. As a preliminary step, each device needs to be manually registered to the online platform. This produces a configuration file which must then be embedded into the image before flashing the device. Clearly, this process is cumbersome when dealing with a large number of devices. Moreover, it can easily lead to configuration mistakes such as flashing devices with the wrong image.

Instead, the proposed solution employs a configuration server which holds the configuration of all devices. Devices are provisioned with a generic OS image, which can be easily automated and is less prone to error. The configuration server is deployed at a known location and devices fetch the configuration file at startup. After the configuration is fetched, devices automatically form a swarm and also register themselves to the management portal. This automates the two steps of configuring devices and registering them online. It is still the case that a specific configuration for each device has to be generated and stored in the configuration server, however, this step does not require interaction with the physical devices and can be easily automated too.

Remote Management

Resin.io includes several features that facilitate remote management of devices: it allows deployment and updates of running applications; it provides access to application logs through the online panel; and it allows connecting via SSH directly into containers, which can be useful for debugging purposes. Each device connected to the platform can be monitored and even rebooted remotely. However, there is little control over the Docker Engine itself, as well as over the settings with which containers are run on devices.

The proposed solution allows deployment and updates of running applications, but provides none of the other features such as logs, SSH access or remote reboots. However, it does allow executing any docker command via the user interface of Portainer and its API. This solution achieves a more fine-grained control over the application configuration. Many of the features of Resin.io such as logging and monitoring, could be implemented in the application level on top of the proposed solution as well, by running containers on each device that perform such tasks.

Device Heterogeneity

Both solutions support heterogeneous devices. In the case of Resin.io, the base OS image is provided for over twenty of the most popular embedded boards and it is possible to add support for other boards. However, requiring a custom OS restricts the compatibility with certain boards which might already have a Linux-based operating system and in some cases manufacturers might not be willing to replace the OS which they are already using.

In contrast, the proposed solution works with any device which is running a suitable Docker Engine, regardless of the underlying OS. As a consequence, manufacturers or OS providers can add support for their boards by simply compiling and installing Docker Engine.

Resin.io considers heterogeneity only as the varying types of system boards (e.g., Raspberry Pi, Intel Edison). In contrast, the proposed solution allows modeling the different types of heterogeneity through labels, for instance, the availability of certain sensors or actuators, the deployment location, and the connectivity type.

Application Heterogeneity

Resin.io supports various devices of different hardware configurations. However, applications can only be deployed on devices of the same type (e.g., Raspberry Pis). Indeed, Resin.io does not allow deploying applications on mixed groups of heterogeneous devices. For instance, to deploy the same application on several devices of different types, it must be done once for each type of device. This increases the management complexity, as each application has to be separately configured, provisioned and monitored. Although it is sensible to target specific device types for certain applications, for others there is barely any relevance of the underlying hardware, such as in the case of system logging and monitoring.

The proposed solution, instead, allows deploying services across heterogeneous devices. A service can be deployed on devices of varying architectures and can be managed (e.g., provisioned, scaled, updated) jointly, instead of doing it separately for each device type. The framework ensures that the correct image for the device architecture is used by means of multi-architecture images. Moreover, sets of devices can be targeted by means of placement constraints, which enable refining the target group not only by device type but also by other characteristics such as the availability of sensors and actuators.

Multiple Applications

Resin.io allows only a single user application to run on each device. Containers are employed to simplify provisioning devices with preconfigured environments and to perform updates. However, the isolation provided by containers is not leveraged. Moreover, any device connected to their platform is required to run a specific application, there are no idle or stand-by devices that could be dynamically allocated.

In contrast, the proposed solution allows deploying several services (i.e., containers) per device. The containers can be isolated from each other or they can interact by means of networks and shared files. The swarm managers allocate and de-allocate containers on each device as services are scaled up and down. The swarm can be seen as a pool of resources that can be dynamically accessed, providing greater computational flexibility. Replication modes and placement constraints further enable deployment of a certain number of container replicas, in some cases even on the same device.

Application Composition

Resin.io provides no support for composing applications which are deployed on devices. Each device is handled as a standalone entity. Networks, service discovery and load-balancing must be explicitly handled by the developer if two running applications have to be connected together. This design favors cloud-based architectures, with cloud endpoints deployed at known locations. Devices do local processing and communicate with cloud servers, but communication among end-devices is not straightforward.

The proposed solution enables application composition by means of virtual overlay networks. Services can be attached to multiple isolated networks, and the containers that make up the services can interact just as physical hosts a local network. Docker Swarm handles name resolution and load-balancing. Running applications can call other applications without any knowledge of container or node IP addresses. Moreover, as the swarm configuration changes, i.e., containers are started or stopped, services continue to be available transparently.

5.2 Image Size

In the previous chapter, a method was presented for generating compact container images through multi-stage builds. In this section, the images produced with this technique are compared with images generated through standard builds to measure

the corresponding reduction in size. For this purpose, a sample application (i.e., a static HTTP endpoint) was developed in four languages, namely Node.js, Go, and C. Python and Node.js were chosen as they have become very popular languages for cloud-based applications and could foster the development community in fog computing. Go was chosen as it is the language in which Docker is written, and parts of the Docker code can be directly imported. C was chosen as it is a language commonly used for developing embedded systems applications.

For each programming language, two images were generated by using standard builds and one by using multi-stage builds. In the case of the standard builds two different base images were tested: the official images (i.e., `python`, `node`, `golang`, `gcc`), which contain many programs and tools useful for debugging, but are large in size; and a smaller-sized official image based on alpine Linux (i.e., `python:2.7-alpine`, `node:alpine`, `golang:alpine`), a lightweight distribution. There is no official alpine image for `gcc`, so one was built by deriving from the `alpine` image and adding the compiler as well as standard libraries.

Multi-stage builds employed two stages to generate compact images. In the case of Python and Node.js, the first stage fetches the application dependencies and the second stage copies the application, its dependencies, the Node.js interpreter and runtime libraries into the final image. In the case of Go and C, the first stage compiles the application with statically-linked libraries and the second stage includes only the executable binary.

Results are presented in Table 3. Standard builds using standard base images incur in a large overhead, as they include large Linux distributions with many software packages. They are useful in early stages of development and prototyping, as they have many tools useful for debugging, but they should not be used for provisioning end-devices. Alpine-based images achieved a significant reduction in size, yet they are still considerably large for over-the-air provisioning when bandwidth is limited or costly. Images built through multi-stage builds offer a tremendous size reduction.

Method	**Python**	**Node.js**	**Go**	**C**
Standard build (regular image)	677 Mb	667 Mb	702 Mb	1.64 Gb
Standard build (alpine image)	182 Mb	68.3 Mb	274 Mb	102 Mb
Multi-stage build	162 Mb	35 Mb	3.9 Mb	950 Kb

Table 3: Size of resulting images when generated through standard and multi-stage builds

The results indicate that compiled languages produce more compact images than interpreted languages. Whereas the compiled programs include only the code and linked libraries that are needed to run, the interpreted programs require the interpreter to be a part of the produced image. These interpreters include unused features, such as interactive environments and libraries which are not required for the

specific application being executed. Smaller versions of interpreters for embedded systems are available, but in many cases compatibility is not completely guaranteed.

It is important to consider how Docker images are built when analyzing these results. Images are made up from a stack of read-only layers that can be reused. When an image is updated, only the new or modified layers need to be retrieved. Although the first provisioning of an application on a device requires the device to download the entire image, successive updates are much lighter. In this case, the performance overhead incurred by using, for instance, a Python image, is not always as significant as Table 3 indicates. If developers ensure that the required base images are available in the devices before-hand – e.g., by incorporating them into the base operating system – then run-time overhead is substantially reduced. Likewise, if all deployed applications are developed using the same language and libraries, based layers are shared among them. However, in the case of performance-critical applications, compiled languages like C are preferable as overhead is minimal even when an image is deployed for the first time.

6 Conclusion

This work presented a distributed computing framework for heterogeneous embedded devices based on software containers. Devices with different characteristics – such as sensors, actuators and system resources – form a cluster where distributed applications are deployed. By running applications directly on end-devices, latency of performance-critical applications is reduced and the resource utilization of IoT devices is maximized. By using containers, applications are easily deployed and scaled on end-devices.

Existing tools were employed and extended with own tools to enable the creation and management of clusters of devices. A model architecture was proposed, which simplifies cluster initialization and management. This architecture supports the interaction between cloud- and fog-based resources.

Docker Swarm was employed to set up clusters of devices (swarms) and to orchestrate containers on them. This container orchestration framework includes many features which facilitate distributed computing such as built-in service discovery, load-balancing, and application composition. By modeling device heterogeneity through node metadata, it is possible to target and deploy applications on specific groups of devices.

Remote management of swarms was achieved through Portainer, a web-based graphical user interface for managing hosts running Docker. Through this interface it is possible to deploy and configure services on multiple swarms. A thin software layer was developed, which enables a cloud-based Portainer installation to connect to swarm managers in the fog.

A configuration server was designed which automates cluster initialization. When each device boots up, a container is run on it which fetches a device-specific configuration file from the configuration server in the cloud. Through the guidelines contained in this file devices are capable of forming a swarm automatically. This eliminates manual configuration steps and simplifies deployment of vast amounts of devices.

A method for building compact multi-architecture container images was described as well. These images are compatible with heterogeneous and resource-constrained devices. The proposed technique achieved a significant reduction in the image size, thus, enabling faster provisioning of devices and less bandwidth usage.

This work was evaluated by comparing against an existing service for container-based IoT device provisioning. The comparison indicated that the proposed framework is easier to set up, has more extensive support for device and application heterogeneity, allows for a more fine-grained control over how applications are deployed, and supports parallel processing and application composition, whereas the existing service does not.

The concepts presented in this work can be expanded and further research can be conducted. In this sense, some interesting directions for future work are the following:

- Measuring the ideal size of swarms of devices, and the effect of swarm size on performance.

- Implementing the configuration server designed in this work, by taking into account security considerations as well.
- Evaluating other strategies for automatic swarm discovery and formation. For instance, devices on the same local network could share information via a service discovery protocol and form a swarm without contacting a cloud-server.
- Developing logging and monitoring functionality such as that provided by Resin.io. This could be realized at the application level, on top of the proposed framework.
- Evaluating the different ways in which physical resources (i.e., sensors, actuators) can be accessed from containers. Containers on a host are isolated environments, but physical resources are shared which can lead to containers with conflicting over the same resource.
- Studying mechanisms for installing Docker Engine on IoT devices, which have varying Linux-based host operating systems. Templates could be provided to incorporate Docker binaries into embedded Linux distributions[8].
- Reliability vs intermittent/noisy network conditions and device failures/malfunctions.

[8] https://www.yoctoproject.org/

References

[1] Best practices for writing Dockerfiles. https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/, 2017. Accessed: 2017-09-07.

[2] Chapter 1. introduction to control groups (cgroups). https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html, 2017. Accessed: 2017-07-13.

[3] Docker hub. https://hub.docker.com/, 2017. Accessed: 2017-07-16.

[4] Docker overview. https://docs.docker.com/engine/docker-overview/, 2017. Accessed: 2017-09-07.

[5] Docker reference architecture: Designing scalable, portable Docker container networks. https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks, 2017. Accessed: 2017-09-10.

[6] Docker registry HTTP API V2. https://docs.docker.com/registry/spec/api/, 2017. Accessed: 2017-08-15.

[7] How nodes work. https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/, 2017. Accessed: 2017-09-09.

[8] IPVS software - advanced layer-4 switching. http://www.linuxvirtualserver.org/software/ipvs.html, 2017. Accessed: 2017-09-10.

[9] manifest-tool. https://github.com/estesp/manifest-tool, 2017. Accessed: 2017-08-18.

[10] Namespaces - overview of linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, 2017. Accessed: 2017-07-13.

[11] svipc - system v interprocess communication mechanisms. http://man7.org/linux/man-pages/man7/svipc.7.html, 2017. Accessed: 2017-07-16.

[12] Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/, 2017. Accessed: 2017-07-17.

[13] Use multi-stage builds. https://docs.docker.com/engine/userguide/eng-image/multistage-build/, 2017. Accessed: 2017-09-03.

[14] What is a container. https://www.docker.com/what-container, 2017. Accessed: 2017-08-15.

[15] J.P. Vasseur A. Dunkels. IP for smart objetcs, internet protocol for smart objects (IPSO) allience, white paper #1. September 2008.

[16] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[17] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 16. ACM, 2017.

[18] David Bernstein. Containers and cloud: From LXC to Đocker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[19] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. pages 13–16, 2012.

[20] Maarten Botterman. Internet of things: an early reality of the future internet. In *Workshop Report, European Commission Information Society and Media*, 2009.

[21] Mung Chiang and Tao Zhang. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[22] Mario Di Francesco, Na Li, Long Cheng, Mayank Raj, and Sajal K Das. A framework for multimodal sensing in heterogeneous and multimedia wireless sensor networks. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*, pages 1–3. IEEE, 2011.

[23] R. Draves. Default address selection for internet protocol version 6 (IPv6). RFC 3484, RFC Editor, February 2003.

[24] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Rasool Fakoor, Mayank Raj, Azade Nazi, Mario Di Francesco, and Sajal K Das. An integrated cloud-based framework for mobile phone sensing. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 47–52. ACM, 2012.

[26] Viktor Farcic. *The DevOps 2.1 Toolkit. Docker Swarm.* LearnPub, 1 2017.

[27] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[28] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. AAI9980887.

[29] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond.

[30] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. In *Proceedings of the 6th International Conference on Networked Sensing Systems*, INSS'09, pages 196–199, Piscataway, NJ, USA, 2009. IEEE Press.

[31] Asad Javed. Container-based IoT sensor node on raspberry Pi and the Kubernetes cluster framework. Master's thesis, Aalto University, 2016.

[32] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[33] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[34] Friedemann Mattern and Christian Floerkemeier. *From the Internet of Computers to the Internet of Things*, pages 242–259. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[35] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[36] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.

[37] Roberto Morabito, Riccardo Petrolo, Valeria Loscrí, and Nathalie Mitton. Enabling a lightweight edge gateway-as-a-service for the internet of things. In *Network of the Future (NOF), 2016 7th International Conference on the*, pages 1–5. IEEE, 2016.

[38] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[39] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[40] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR)*, 45(2):17, 2013.

[41] Thi Anh Mai Phan, Jukka K Nurminen, and Mario Di Francesco. Cloud databases for internet-of-things data. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE*, pages 117–124. IEEE, 2014.

[42] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[43] Flávio Ramalho and Augusto Neto. Virtualization at the network edge: A performance comparison. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016 IEEE 17th International Symposium on A*, pages 1–6. IEEE, 2016.

[44] Mohit Sethi, Pranvera Kortoci, Mario Di Francesco, and Tuomas Aura. Secure and low-power authentication for resource-constrained devices. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 30–36. IEEE, 2015.

[45] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[46] International Telecommunication Union. Ict facts and figures 2017. http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf, July 2017.

[47] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A scalable framework for provisioning large-scale IoT deployments. *ACM Transactions on Internet Technology (TOIT)*, 16(2):11, 2016.

[48] Marcelo Yannuzzi, R Milito, René Serral-Gracià, D Montero, and Mario Nemirovsky. Key ingredients in an IoT recipe: Fog computing, cloud computing, and more fog computing. In *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2014 IEEE 19th International Workshop on*, pages 325–329. IEEE, 2014.